ISTQB® Certified Tester
Advanced Level

Agile Technical Tester (CTAL-ATT)

Lehrplan

Version 2019 (09.12.2019)

Deutschsprachige Ausgabe
Herausgegeben durch das German Testing Board e. V.
in Zusammenarbeit
mit dem Austrian Testing Board
und dem Swiss Testing Board

Übersetzung des englischsprachigen Lehrplans des International Software Testing Qualifications Board (ISTQB®), Originaltitel: Certified Tester, Advanced Level Syllabus, Agile Technical Tester (ATT), Version 1.1.

Urheberschutzvermerk

Änderungsübersicht der deutschsprachigen Ausgabe

Version	Datum	Bemerkung
01	07.01.2020	Erster Entwurf
02	14.02.2020	Internes Review
03	17.02.2020	Reviewkommentare eingearbeitet
0.4	12.11.2020	Reviewergebnisse der GTB AG Agile Tester eingearbeitet
1.0	27.11.2020	Release Version

Inhaltsverzeichnis

Danksagung

Der englischsprachige Lehrplan wurde vom Team der Agile Working Group des International Software Testing Qualifications Board unter der Leitung von Rex Black (Vorsitzender), Michaël Pilaeten (stellvertretender Vorsitzender und Vorsitzender a. i.) und Renzo Cerquozzi (Product Owner) erstellt.

Das Advanced Level Agile-Team dankt dem Review-Team und den Nationalen Boards für ihre Vorschläge und Beiträge.

Zum Zeitpunkt der Fertigstellung des Lehrplans Advanced Level – Agile Technical Tester hatte die Working-Group folgende Mitglieder: Michaël Pilaeten (Vorsitz a. i.), Renzo Cerquozzi (Product Owner), Alon Linetzki (Working-Group Marketing), Leanne Howard (Working-Group Glossar) und Klaus Skafte (Working-Group Exam).

Autoren: Leo van der Aalst, Renzo Cerquozzi, Bertrand Cornanguer, István Forgács, Jani Haukinen, Noam Kfir, Sammy Kolluru, Alon Linetzki, Tilo Linz, Michaël Pilaeten, Marie Walsh.

Interne Reviewer des englischsprachigen Lehrplans: Michael Arefi, Vojtěch Barta, Renzo Cerquozzi, Graham Bath, Laurent Bouhier, Anders Claesson, Alessandro Collino, David Janota, David Evans, Leanne Howard, Matthias Hamburg, Kari Kakkonen, Tor Kjetil Moseng, Meile Posthuma, Salvatore Reale, Marko Rytkönen, Sarah Savoy, Klaus Skafte, Mike Smith, Chris van Bael.

Erstellung und Überprüfung der englischsprachigen ‚Sample Questions': Armin Born, Renzo Cerquozzi, Alon Linetzki, Tilo Linz, Jamie Mitchell, Jani Haukinen, Tobias Horn, Chris van Bael, Suruchi Varshney.

Das Team dankt auch den folgenden Personen aus den Nationalen Boards und der Gemeinschaft agiler Experten, die an der Überprüfung, Kommentierung und Abstimmung des englischsprachigen Lehrplans teilgenommen haben: Adam Roman, Armin Beer, Beata Karpinska, Chris Van Bael, Erwin Engelsma, Giancarlo Tomasig, Gary Mogyorodi, Ingvar Nordström, Jana Gierloff, Jörn Münzel, Jurian van de Laar, Kari Kakkonen, Laurent Bouhier, Marko Rytkönen, Martin Klonk, Matthias Hamburg, Meile Posthuma, Paul Weymouth, R. Green, Richard Seidl, Rik Marselis, Stephanie Ulrich, Stephanie van Dijck, Tal Pe'er, Tilo Linz, Veronica Seghieri und Wim Decoutere.

Der englischsprachige Lehrplan wurde von der General Assembly des ISTQB® am 18. Oktober 2019 formell zur Veröffentlichung genehmigt.

Das German Testing Board (GTB), das Swiss Testing Board (STB) und das Austrian Testing Board (ATB) danken der T-Systems International GmbH für die initiale Übersetzung des Lehrplanes, insbesondere Andrea Keintzel und Dr. Birgit Dressler, sowie der Bereitschaft, diese Übersetzung zur Verfügung zu stellen.

Weiterhin bedanken sich die Boards beim Reviewteam der deutschsprachigen Fassung: Andrea Keintzel, Baris Gueldali, Jörn Münzel, Markus Zaar, Martin Klonk, Michael Heller, Stephanie Ulrich und Tilo Linz (Leitung).

0 Einführung

0.1 Zweck dieses Lehrplans

Dieser Lehrplan bildet die Grundlage für die Qualifikation als Agile Technical Tester auf der Aufbaustufe Advanced Level des Softwaretestausbildungsprogramms des International Software Testing Qualifications Board (im Folgenden ISTQB® genannt). Das German Testing Board e. V. (im Folgenden GTB® genannt) hat diesen Lehrplan in Zusammenarbeit mit dem Austrian Testing Board (ATB) und dem Swiss Testing Board (STB) in die deutsche Sprache übersetzt. Das GTB® und ISTQB® stellen den Lehrplan folgenden Adressaten zur Verfügung:

- Nationalen Mitglieds-Boards, die den Lehrplan in ihre Sprache(n) übersetzen und Seminaranbieter akkreditieren dürfen. Nationale Mitglieds-Boards dürfen den Lehrplan an die Anforderungen ihrer nationalen Sprache anpassen und Referenzen auf lokale Veröffentlichungen hinzufügen.
- Zertifizierungsstellen zur Ableitung von Prüfungsfragen in ihrer nationalen Sprache, die an die Lernziele dieses Lehrplans angepasst sind.
- Seminaranbietern zur Erstellung von Kursmaterialien und zur Bestimmung angemessener Lehrmethoden.
- Zertifizierungskandidaten zur Vorbereitung auf die Zertifizierungsprüfung (entweder als Teil eines Seminars oder unabhängig davon).

Der internationalen Software- und Systementwicklungs-Community zur Förderung des Berufsbildes des Software- und Systemtesters und als Grundlage für Bücher und Fachartikel. Das GTB® und ISTQB® können die Nutzung dieses Lehrplans auch anderen Personenkreisen oder Institutionen für andere Zwecke erlauben, sofern diese vorab eine entsprechende schriftliche Genehmigung erfragen und erhalten.

0.2 Übersicht

Das Übersichtsdokument zum ISTQB® Advanced Level Agile Technical Tester [ISTQB_ATT_OVIEW] enthält die folgenden Informationen:

- Geschäftlicher Nutzen des Lehrplans:
    1. Anwenden agiler Verfahren, um sicherzustellen, dass die Tests dadurch eine angemessene Abdeckung bieten
    2. Definieren testbarer Anforderungen innerhalb des agilen Teams
    3. Konzipieren und Anwenden verschiedener agiler Testansätze unter Verwendung geeigneter Verfahren
    4. Unterstützen und Mitwirken bei Testautomatisierungsaktivitäten in einem agilen Projekt

5. Unterstützen im agilen Team bei Continuous Integration (kontinuierlichen Integration)
6. Unterstützen im agilen Team bei Continuous Delivery und Deployment (kontinuierliche Auslieferung und Bereitstellung/Inbetriebnahme)
7. Kennenlernen von Service-Virtualisierungskonzepten
8. Mit anderen Teammitgliedern zusammenarbeiten und Informationen austauschen unter Anwendung effektiver Kommunikationsstile und -kanäle.

- Eine Matrix, die die Nachverfolgbarkeit zwischen geschäftlichem Nutzen und Lernzielen darstellt
- Zusammenfassung des Lehrplans
- Anhänge zu Referenzen des Lehrplans: Bücher, Artikel, Links, siehe auch Kapitel 5

0.3 Prüfungsrelevante Lernziele

Die Lernziele unterstützen die geschäftlichen Ziele und dienen zur Ausarbeitung der Prüfung für die Zertifizierung zum Advanced Level Agile Technical Tester. Den einzelnen Lernzielen ist jeweils eine kognitive Wissensstufe (K-Stufe) zugeordnet.

Die K-Stufe (bzw. kognitive Wissensstufe) dient dazu, Lernziele gemäß der überarbeiteten Taxonomie von Bloom [Anderson01] zu klassifizieren. Das ISTQB® verwendet diese Taxonomie bei der Erstellung der Prüfungen zu den Lehrplänen.

Dieser Lehrplan berücksichtigt folgende der insgesamt sechs möglichen kognitiven Wissensstufen:

Stufe	**Kennwort**	**Beschreibung**
1	Kennen	Der Lernende kann einen Begriff oder ein Konzept erkennen, abrufen und sich daran erinnern.
2	Verstehen	Der Lernende kann Aussagen zum Thema begründen oder erläutern.
3	Anwenden	Der Lernende kann erworbenes Wissen auf gegebene neue Situationen übertragen oder zur Problemlösung anwenden.
4	Analysieren	Der Lernende kann Informationen mit Bezug auf eine Vorgehensweise oder ein Verfahren zum besseren Verständnis in ihre einzelnen Bestandteile aufgliedern sowie zwischen Fakten und Rückschlüssen unterscheiden.

Jedes Lernziel wird gemäß seiner kognitiven Stufe geprüft. Die relevanten Lernziele der kognitiven Stufen K2 und höher werden immer zu Beginn des jeweiligen Kapitels angegeben. Darüber hinaus sind für alle Schlüsselbegriffe, die zu Beginn des jeweiligen Kapitels gelistet werden, die jeweiligen Definitionen aus dem GTB/ISTQB Standardglossar der Testbegriffe [GTB-GLOSSAR] entsprechend der kognitiven Stufe K1 prüfungsrelevant.

Allgemein gilt, dass der gesamte Inhalt des vorliegenden Lehrplans entsprechend der kognitiven Stufe K1 geprüft werden kann. Dies bedeutet, dass die Prüfungskandidaten Begriffe und Inhalte erkennen, sich an sie erinnern und sie wiedergeben können.

0.4 Die Prüfung zum Advanced Level Agile Technical Tester

Die Zertifizierungsprüfung für den Advanced Level Agile Technical Tester basiert auf diesem Lehrplan. Zur Beantwortung einer Prüfungsfrage kann Wissen aus mehreren Abschnitten dieses Lehrplans erforderlich sein. Alle Abschnitte dieses Lehrplans sind prüfungsrelevant, außer der Einführung und der Anhänge. Im Lehrplan sind Standards, Fachbücher und andere ISTQB®-Lehrpläne als Referenzen genannt; deren Inhalt ist jedoch nicht über das hinaus prüfungsrelevant, was im vorliegenden Lehrplan in zusammengefasster Form enthalten ist.

Das Format der Prüfung ist Multiple Choice.

Prüfungen können als Teil eines akkreditierten Seminars oder unabhängig davon (z. B. bei einer Zertifizierungsstelle oder in einer öffentlichen Prüfung) abgelegt werden. Die Teilnahme an einem akkreditierten Seminar stellt keine Voraussetzung für das Ablegen der Prüfung dar.

0.5 Akkreditierung

Nationale ISTQB®-Mitglieds-Boards können Seminaranbieter akkreditieren, deren Seminarmaterialen diesem Lehrplan entsprechen.

Die Seminaranbieter sollten sich von ihrem nationalen Board (in Deutschland: German Testing Board e. V.; in der Schweiz: Swiss Testing Board; in Österreich: Austrian Testing Board) oder von der Stelle, die die Akkreditierungen durchführt, die entsprechenden Akkreditierungsrichtlinien einholen. Ein akkreditiertes Seminar ist als lehrplankonform anerkannt, und es darf im Rahmen des Seminars eine ISTQB®-Prüfung abgelegt werden.

0.6 Aufbau des Lehrplans

Der Lehrplan besteht aus vier Kapiteln mit prüfungsrelevanten Inhalten.

Die Hauptüberschrift für jedes Kapitel spezifiziert jeweils die Unterrichtszeit für das Kapitel; eine weitere Aufgliederung der Zeitangabe ist nicht vorgesehen.

Für akkreditierte Seminare erfordert der Lehrplan eine Gesamtunterrichtszeit von mindestens **16** Stunden, die sich wie folgt auf die vier Kapitel verteilen:

- Kapitel 1 – Requirements Engineering: 180 Minuten
- Kapitel 2 – Testen im agilen Umfeld: 540 Minuten
- Kapitel 3 – Testautomatisierung: 135 Minuten
- Kapitel 4 – Deployment und Delivery (Bereitstellung/Inbetriebnahme und Auslieferung): 105 Minuten

Bezüglich der Mindestunterrichtszeiten für die jeweiligen kognitiven Stufen der Lernziele siehe [ISTQB_ATT_OVIEW].

1 Requirements Engineering – 180 Minuten

Schlüsselbegriffe

Abnahmekriterien, Epic, User-Story

Lernziele für „Requirements Engineering“

ATT-1.x (K1) Schlüsselbegriffe

1.1 Requirements-Engineering-Verfahren

ATT-1.1.1-1 (K4) User-Storys und Epics mit Hilfe von Requirements-Engineering-Verfahren analysieren können

ATT-1.1.1-2 (K2) Requirements-Engineering-Verfahren beschreiben und wie sie Tester unterstützen können

ATT-1.1.2-1 (K4) Testbare Abnahmekriterien für eine bestimmte User-Story mit Hilfe von Requirements-Engineering-Verfahren und den Testverfahren erstellen und bewerten können

ATT-1.1.2-2 (K2) Die Ermittlungstechniken beschreiben können

1.1 Requirements-Engineering-Verfahren

Die Anwendung von Requirements-Engineering-Verfahren ermöglicht es agilen Teams, die User-Storys (siehe [AgileFoundationExt]) und Epics (siehe [AgileFoundationExt]) zu verfeinern, einen Kontext hinzuzufügen, Auswirkungen und Abhängigkeiten zu berücksichtigen und mögliche Lücken wie z. B. fehlende nicht-funktionale Anforderungen zu identifizieren.

Obwohl die Mehrheit der in diesem Abschnitt behandelten Requirements-Engineering-Verfahren aus dem Bereich der traditionellen Entwicklungsansätze stammt, sind diese auch in der agilen Entwicklung erfolgreich anwendbar.

Im Allgemeinen sind in herkömmlichen Projekten die Aktivitäten und Verfahren des Requirements Engineering formalisiert, sie werden sequentiell durchgeführt und liegen in der Verantwortung von dedizierten Personen wie Businessanalysten, fachliche Analysten, technische Architekten, Enterprise-Architekten und Prozessanalysten.

Im Gegensatz dazu werden in agilen Projekten die Requirements-Engineering-Verfahren während der gesamten Projektlaufzeit und innerhalb jeder Iteration mit einem weniger formalen Ansatz angewendet. Die Requirements-Engineering-Verfahren werden häufiger durch kontinuierliche Feedbackschleifen und von allen Mitgliedern des agilen Teams angewendet, nicht nur durch die dedizierten Businessanalysten oder Product Owner.

1.1.1 Analyse von User-Storys und Epics mit Requirements-Engineering-Verfahren

Um als Tester bei der Klärung (und möglicherweise Verbesserung) von User-Storys, Epics und anderen agilen Anforderungen mitzuwirken, ist es notwendig, die verschiedenen Requirements-Engineering-Verfahren, die dies unterstützen, zu kennen, zu verstehen sowie auswählen und anwenden zu können.

Beispiele für solche Verfahren oder Techniken sind Storyboards, Story-Mapping, Personas, Diagramme und Anwendungsfälle (Use-Cases):

- **Storyboards:** Ein Storyboard (nicht zu verwechseln mit agilen Taskboards oder agilen User-Storyboards) liefert eine visuelle Darstellung des Systems. Die Verwendung von Storyboards unterstützt den Tester bei den folgenden Tätigkeiten:
  - Erkennen der Intension hinter den User-Storys und der Gesamtsicht auf die ‚overall Story', die den Kontext liefert und es ermöglicht, den funktionalen Ablauf im System erkennen und mögliche Lücken in der Logik identifizieren zu können.
  - Visualisieren von Gruppen von User-Storys, die sich auf einen gemeinsamen Bereich des Systems (Themes) beziehen und für die Aufnahme in derselben Iteration in Betracht gezogen werden können, da sie wahrscheinlich denselben Teil des Codes betreffen werden.
  - Mitwirken am Story-Mapping und bei der Priorisierung von Epics und den zugehörigen User-Storys im Product-Backlog.
  - Mitwirken beim Identifizieren von Abnahmekriterien für User-Storys und Epics.
  - Mitwirken an der Auswahl der richtigen Testvorgehensweise basierend auf der visuellen Darstellung des Systemdesigns im Storyboard.
  - In Verbindung mit Story-Mapping mitwirken am Priorisieren der Tests und am Identifizieren des Bedarfs an Stubs (Platzhaltern), Treibern und/oder Mocks.
- **Story-Mapping:** Story-Mapping (oder User-Story-Mapping) ist eine Technik, mit deren Hilfe zwei unabhängige Dimensionen verwendet werden, um User-Storys zu ordnen. Die horizontale Achse der Story-Map repräsentiert die Prioritätenrangfolge jeder User-Story, während die vertikale Achse die erreichte Detailtiefe (bzw. zunehmende Verfeinerung) in der Umsetzung darstellt. Die Verwendung von Story-Mapping kann den Tester bei Folgendem unterstützen:
  - Bestimmen der grundlegendsten Funktionalitäten eines Systems zur Ableitung des Smoke-Tests.
  - Identifizieren der Reihenfolge der Funktionalitäten, um die Testprioritäten zu ermitteln.
  - Visualisieren des Systemumfangs.
  - Bestimmen der Risikostufe jeder User-Story.

- **Personas:** Personas werden verwendet, um fiktive Charaktere oder Archetypen zu definieren, die veranschaulichen, wie typische Benutzer mit dem System interagieren. Die Verwendung von Personas kann den Tester bei Folgendem unterstützen:
  - Aufdecken von Lücken in den User-Storys durch das Identifizieren von verschiedenen Arten von Benutzertypen, die das System nutzen können.
  - Identifizieren von Inkonsistenzen in User-Storys dadurch, wie ein bestimmter Benutzertyp das System im Vergleich zu anderen Benutzertypen nutzt.
  - Ermitteln von Abnahmekriterien für die User-Storys.
  - Entdecken zusätzlicher Testpfade beim explorativen Testen.
  - Identifizieren von Testbedingungen, insbesondere derjenigen, die sich auf bestimmte Benutzergruppen beziehen um auf diese Weise eine ausreichende Abdeckung der Benutzergruppen und den Test der Unterschiede zwischen den Benutzergruppen sicherzustellen.
- **Diagramme:** Diagramme wie Entity-Relationship-Diagramme, Klassendiagramme und (andere) UML-Diagramme können die Struktur oder den Datenfluss sowie die funktionalen Attribute oder das Verhalten des Systems darstellen. Sie können verwendet werden, um Lücken in der Systemfunktionalität zu identifizieren.
- **Anwendungsfall (Use-Case):** Anwendungsfälle (Diagramme und Spezifikationen) (siehe [Foundation]) können den Tester bei Folgendem unterstützen:
  - Sicherstellen, dass die User-Storys testbar sind und eine angemessene Größe haben.
  - Entscheiden, ob die User-Storys verfeinert oder zerlegt werden müssen.
  - Aufdecken von vergessenen Stakeholdern.
  - Identifizieren von Schnittstellen und Integrationspunkten, die beim Testentwurf zu berücksichtigen sind.
  - Erkennen der Beziehungen zwischen Epic und User-Storys, um zu prüfen, dass ein Epic vollständig durch User-Storys abgedeckt ist bzw. dass zu einem Epic keine User-Storys fehlen.

1.1.2 Identifizierung von Abnahmekriterien durch Requirements-Engineering-Verfahren und Testverfahren

Requirements Engineering ist ein Prozess, der aus den folgenden Schritten besteht:

- **Ermitteln:** Aufdecken, Verstehen, Dokumentieren und Überprüfen der Benutzerbedürfnisse und der Randbedingungen für das System. Ermittlungstechniken sollten zur Ableitung, Bewertung und Verbesserung der Abnahmekriterien verwendet werden.
- **Dokumentieren:** Eindeutige und verständliche Dokumentation der Benutzerbedürfnisse und Randbedingungen. User-Storys und Abnahmekriterien

sollten mit einem Detaillierungsgrad dokumentiert werden, welcher der Einhaltung der Prinzipien des Agilen Manifestes durch das Team entspricht. Die Art der Dokumentation hängt von dem im Team mit den Stakeholdern gewählten Ansatz ab. Abnahmekriterien können unter Verwendung natürlicher Sprache, unter Verwendung von Modellen (z. B. Zustandsübergangsdiagramme) oder unter Verwendung von Beispielen dokumentiert werden.

- **Abstimmen und Bestätigen:** Bei jeder User-Story haben mehrere Stakeholder möglicherweise unterschiedliche Ansichten und Wünsche. Da diese Ansichten und Wünsche inkonsistent oder sogar widersprüchlich sein können, kann dies auch auf die Abnahmekriterien der einzelnen Stakeholder zutreffen. Jeder dieser Konflikte sollte zwischen allen betroffenen Stakeholdern identifiziert, abgestimmt und gelöst werden. Jeder vergessene oder ungelöste Konflikt könnte den Erfolg des Projekts gefährden. Am Ende dieses Schritts ist der Inhalt jeder User-Story durch die betroffenen Stakeholder bestätigt (z. B. als Definition of Ready).
- **Verwalten:** Im Verlauf von Projekten können sich Meinungen und Gegebenheiten ändern. Auch wenn die Annahmekriterien ordnungsgemäß ermittelt, dokumentiert, abgestimmt und bestätigt wurden, können sich die Abnahmekriterien noch ändern. Aufgrund des Änderungspotenzials sollten User-Storys mit Hilfe guter Konfigurations- und Change-Management-Prozesse verwaltet werden.

Zur Identifizierung der Abnahmekriterien stehen dem Tester verschiedene Ermittlungstechniken zur Verfügung, u. a.:

- **Quantitative Fragebögen:** Die Verwendung quantitativer Daten, die aus geschlossenen Fragen gewonnen werden, ist eine ausgezeichnete Möglichkeit, eindeutige Vergleiche zwischen verschiedenen Datenpunkten anzustellen. Die quantitative Befragung liefert häufig Datenwerte, die in eine numerische Entscheidbarkeit für ein Abnahmekriterium einbezogen werden können. Der quantitative Fragebogen kann als Ermittlungstechnik bei einer großen Anzahl von Stakeholdern und insbesondere für nicht-funktionale Abnahmekriterien verwendet werden.
- **Qualitative Fragebögen:** Offene Fragen sind eine äußerst effektive Möglichkeit, der quantitativen Analyse mehr Qualität zu verleihen. Offene Fragen werden idealerweise als nachfolgende Ergänzung für wesentliche Fragen verwendet. Hierdurch können zusätzliche Informationen entstehen, für die neue User-Storys erstellt oder die zu vorhandenen User-Storys hinzugefügt werden müssen. Der qualitative Fragebogen kann als Ermittlungstechnik bei einer geringeren Anzahl von Stakeholdern verwendet werden – da die Durchführung und Auswertung mehr Zeit in Anspruch nimmt – und eignet sich für die Definition funktionaler Abnahmekriterien.
- **Qualitatives Interview:** Das qualitative Interview ist flexibler als eine quantitative Befragung und dient hauptsächlich dazu, Informationen über Hintergründe, Zusammenhänge und Ursachen bzw. Beweggründe zu erhalten. Es ist

unwahrscheinlich, dass hierbei konkrete Daten und Werte zurückgeliefert werden, aber aus diesen Antworten können Abnahmekriterien bezüglich des Kontexts einer User-Story abgeleitet werden. Das qualitative Interview kann eine Ergänzung zu jeder Art von Fragebogen sein, um die abgeleiteten Abnahmekriterien zu verfeinern.

Es existieren zahlreiche andere Ermittlungstechniken, vom Bereich der Beobachtungstechniken (z. B. Apprenticing, ‚über die Schulter schauen') über Kreativitätstechniken (z. B. die sechs Denkhüte) hin zu unterstützenden Techniken (z. B. Low-Fi-Prototyping / low fidelity VR prototyping). Der Werkzeugkasten eines Testers an Ermittlungstechniken hat Einfluss auf die Qualität der ermittelten Abnahmekriterien.

Ansätze wie INVEST und SMART (siehe [INVEST]) können zusammen mit Testverfahren wie Äquivalenzklassenbildung, Grenzwertanalyse, Entscheidungstabellen und zustandsbasierte Tests (siehe [Foundation]) ebenfalls verwendet werden um Abnahmekriterien zu identifizieren und festzulegen.

2 Testen im agilen Umfeld – 540 Minuten

Schlüsselbegriffe

Abnahmetestgetriebene Entwicklung (Acceptance Test-Driven Development, ATDD), Spezifikation durch Beispiele (Specification by Example, SBE), Test-Charta, testgetriebene Entwicklung (Test-Driven Development, TDD), verhaltensgetriebene Entwicklung (Behavior-Driven Development, BDD)

Lernziele für das Testen im agilen Umfeld

ATT-2.x (K1) Schlüsselbegriffe

2.1 Agile Entwicklungstechniken und Testverfahren

ATT-2.1.1-1 (K3) Testgetriebene Entwicklung (Test-Driven Development, TDD) im Kontext eines gegebenen Beispiels in einem agilen Projekt anwenden können

ATT-2.1.1-2 (K2) Eigenschaften eines Unittests verstehen können

ATT-2.1.1-3 (K2) Bedeutung des Akronyms FIRST verstehen können

ATT-2.1.2-1 (K3) Verhaltensgetriebene Entwicklung (Behavior-Driven Development, BDD) im Kontext einer gegebenen User-Story in einem agilen Projekt anwenden können

ATT-2.1.2-2 (K2) Verstehen können, wie Richtlinien für die Formulierung von Szenarien eingesetzt werden können

ATT-2.1.3 (K4) Analysieren des Product-Backlog in einem agilen Projekt, um ein Vorgehen zur Einführung einer abnahmetestgetriebenen Entwicklung (Acceptance Test-Driven Development, ATDD) bestimmen zu können

2.2 Erfahrungsbasiertes Testen im agilen Umfeld

ATT-2.2.1-1 (K4) Eine Testvorgehensweise, die automatisierte Tests, erfahrungsbasierte Tests und Black-Box-Tests und die mit Hilfe anderer Vorgehensweisen (einschließlich risikobasierter Tests) für ein vorgegebenes Szenario in einem agilen Projekt entworfen wurde, aus dem Blickwinkel erfahrungsbasierten Testens analysieren und beurteilen können

ATT-2.2.1-2 (K2) Unterschiede zwischen unternehmenskritisch und nicht unternehmenskritisch erläutern können

ATT-2.2.2-1 (K4) User-Storys und Epics analysieren, um eine Test-Charta erstellen zu können

ATT-2.2.2-2 (K2) Den Einsatz von erfahrungsbasierten Testverfahren verstehen können

2.3 Aspekte der Codequalität

ATT-2.3.1-1 (K2) Bedeutung des Refactoring von Testfällen in agilen Projekten verstehen können

ATT-2.3.1-2 (K2) Eine geeignete Aufgabenliste für das Refactoring von Testfällen verstehen können

ATT-2.3.2-1 (K4) Codeanalyse im Rahmen eines Code-Reviews durchführen können, um Fehlerzustände und technische Schulden identifizieren zu können

ATT-2.3.2-2 (K2) Statische Codeanalyse verstehen können

2.1 Agile Entwicklungstechniken und Testverfahren

In der Softwareentwicklung können Fehlerzustände durch schlecht geschriebenen Code und durch nicht eingehaltene Kundenbedürfnisse auftreten. Agile Entwicklungstechniken begegnen diesen Problemen durch Anwendung der Konzepte von testgetriebener Entwicklung (Test-Driven Development, TDD), verhaltensgetriebener Entwicklung (Behavior-Driven Development, BDD) und abnahmetestgetriebener Entwicklung (Acceptance Test-Driven Development, ATDD). TDD zielt auf die richtige Implementierung des Systems ab während BDD und ATDD helfen die Nutzungsqualität (quality in use: Features und Funktionen) zu verbessern.

2.1.1 Testgetriebene Entwicklung (Test-Driven Development, TDD)

Die testgetriebene Entwicklung (TDD) ist eine Softwareentwicklungsmethode, die Design, Test und Codierung in einem schnellen iterativen Prozess kombiniert. TDD verfolgt einen stringenten Test-First-Ansatz, bei dem ein Test erstellt wird, welcher die beabsichtigte Funktionalität des Codes beschreibt, bevor diese Funktionalität codiert wird. Der Test wird ausgeführt bevor der Code selbst programmiert wird, um sicherzustellen, dass der Test fehlschlägt. Sobald der Code programmiert ist, wird der Test erneut ausgeführt und sollte dann bestanden werden.

TDD wird im Allgemeinen als die erste bedeutende Test-First-Programmiermethode angesehen, von welcher sich andere Methoden, wie die verhaltensgetriebene Entwicklung (Behavior-Driven Development, BDD), die abnahmetestgetriebene Entwicklung (Acceptance Test-Driven Development, ATDD) und die Spezifikation durch Beispiele (Specification by Example, SBE), ableiten.

TDD bietet einen evolutionären Ansatz zur Softwareentwicklung, der das Design als einen kontinuierlichen, fortlaufenden Prozess behandelt. Jede Iteration stellt eine kleine Änderung am Programmcode dar und bietet dem Entwickler die Möglichkeit, das Design leicht zu verbessern. Durch die Schritt für Schritt hinzukommenden

Änderungen und sorgfältig durchdachten Designentscheidungen entsteht ein robustes und gut getestetes Design.

TDD-Anwender verwenden eine Reihe von Praktiken und Techniken, um qualitativ hochwertigen Code zu schreiben und die Anhäufung von technischen Schulden (technical debt) zu vermeiden, einschließlich:

- Unittests und einen einheitlichen und vorgegebenen Test-First-Ansatz
- Kurze iterative Zyklen
- Unmittelbares und häufiges Feedback
- Effektiver Einsatz von Werkzeugen, Versionskontrollsystemen (source control) und Continuous Integration (kontinuierliche Integration)
- gezielter Anwendung von Programmierprinzipien und Entwurfsmustern

TDD-Anwender schreiben Unittests, um das erwartete Verhalten des Programmcodes zu überprüfen. Ein Unittest führt eine Funktion unter bestimmten Bedingungen aus und prüft, ob diese ein erwartetes Ergebnis liefert oder nicht. Die Erwartungen werden mit Zusicherungen (assertions) beschrieben, welche überprüfen, ob sich der Systemzustand wie erwartet ändert oder ob ein bestimmtes Verhalten gezeigt wird. Geprüft werden kann z. B.:

- dass eine Funktion eine Berechnung durchführt und ein erwartetes Ergebnis zurückgibt.
- dass eine Funktion den Zustand des Systems auf eine bestimmte Weise modifiziert.
- dass eine Funktion eine andere Funktion auf eine bestimmte Weise aufruft.

Unittests sollten für Entwickler einfach zu realisieren und zu warten sein. Sie sind automatisiert und werden oft in der gleichen Sprache wie der Programmcode geschrieben. Unittests sollten die folgenden Eigenschaften haben:

- **Deterministisch:** Jedes Mal, wenn ein Unittest unter den gleichen Bedingungen durchgeführt wird, sollte dieser die gleichen Ergebnisse zurück liefern.
- **Atomar:** Der Unittest sollte nur die mit ihm verbundene Funktionalität testen.
- **Isoliert:** Ein Unittest sollte möglichst nur den spezifischen Programmcode ausführen, für den er ursprünglich beabsichtigt war. Unittests sollten nicht voneinander abhängig sein und auch Abhängigkeiten gegenüber dem Programmcode sollten nach Möglichkeit vermieden werden.
- **Schnell:** Unittests sollten klein und schnell durchführbar sein, damit sie unmittelbares Feedback liefern. Nach Möglichkeit sollten viele Unittests in einer kurzen Zeitspanne durchführbar sein.

Ein weiteres Akronym für gute Unittests ist FIRST: Fast (schnell), Isolated (isoliert), Repeatable (wiederholbar), Self-Validating (selbstvalidierend), Thorough (gründlich).

Es gibt viele Unittest-Frameworks für viele verschiedene Programmiersprachen. Sie unterscheiden sich in ihren APIs, Ansätzen und Terminologie, aber alle haben einige gemeinsame Elemente. Unabhängig von der Programmiersprache oder dem Unittest-Framework folgt ein Unittest typischerweise folgendem dreistufigen Muster:

- **Einrichten** (arrange/setup): Vorbereiten der Testdurchführungsumgebung. In diesem Schritt können je nach Bedarf Objekte instantiiert, der Systemstatus initialisiert und Daten angelegt werden.
- **Ausführen** (act): Die zu testende Operation ausführen und anschließend das Ergebnis der Operation überprüfen. Dieser obligatorische Schritt kann aus nur einer Codezeile bestehen, welche die zu testende Operation auslöst.
- **Prüfen** (assert): Die tatsächliche Überprüfung der erwarteten Ausgaben oder anderer Nachbedingungen durchführen.

Der iterative Prozess ist der zentrale Grundpfeiler von TDD. Der Zweck jeder Iteration ist es, eine kleine, gezielte, sorgfältig durchdachte Änderung am Programm vorzunehmen. Entsprechend der gebräuchlichen Farben wird der iterative Zyklus häufig als Rot-Grün-Refactoring-Zyklus bezeichnet:

- **Rot:** Einen Test erstellen, der eine bisher nicht realisierte Erwartung beschreibt und fehlschlagen wird. Den Test ausführen, um sicherzustellen, dass dieser fehlschlägt.
- **Grün:** Programmcode schreiben, welcher nur die durch den Test beschriebene Erwartung erfüllt und zum Bestehen des Tests führt. Alle zugehörigen Tests ausführen, um sicherzustellen, dass alle bestanden werden. Wenn einzelne Tests fehlschlagen, dann die notwendigen Änderungen durchführen, bis alle Tests bestanden sind.
- **Refactoring:** Design und Struktur sowohl des Testcodes als auch des Programmcodes verbessern, ohne dass dabei die Funktionalität verändert wird. Gleichzeitig sicherstellen, dass alle zugehörigen Tests weiterhin bestanden werden. Entwickler führen ggf. eine Reihe von Refactoring-Schritten durch, um den Programmcode zu ändern, ohne dabei das Verhalten zu ändern, bis der Programmcode optimiert ist. Die meisten modernen integrierten Entwicklungsumgebungen (Integrated Development Environment, IDE) und viele Code-Editoren können Refactoring automatisch und zuverlässig durchführen.

2.1.2 Verhaltensgetriebene Entwicklung (Behavior-Driven Development, BDD)

Nach ISTQB®-Lehrplan Foundation Level Agile Tester ist die verhaltensgetriebene Entwicklung (BDD) eine Technik, bei der Entwickler, Tester und Fachexperten zusammenarbeiten, um die Anforderungen an ein Softwaresystem zu analysieren, sie in einer gemeinsamen Sprache zu formulieren und automatisch zu verifizieren.

BDD wird im Wesentlichen von folgenden zwei Disziplinen beeinflusst: testgetriebene Entwicklung (Test-Driven Development, TDD) und domänengetriebenes Design (Domain-Driven Design, DDD) (siehe [Evans03]).

BDD beinhaltet viele der Kernprinzipien beider Disziplinen, einschließlich Zusammenarbeit, Design, ubiquitäre (allgemein verwendete, einheitliche) Sprache, automatisierte Testdurchführung, kurze Feedback-Zyklen und vieles mehr. TDD stützt sich auf Unittests, um Realisierungsdetails zu verifizieren, während BDD sich auf ausführbare Szenarien stützt, um Verhaltensweisen und weiteres zu verifizieren. BDD folgt in der Regel einem vorgegebenen Ablauf:

- User-Storys gemeinsam erstellen.
- User-Storys als ausführbare Szenarien und verifizierbare Verhaltensweisen formulieren.
- Verhaltensweisen implementieren und die Szenarien ablaufen lassen, um das Verhalten zu verifizieren.

Teams, die BDD anwenden, extrahieren aus jeder User-Story ein oder mehrere Szenarien und formulieren diese dann als automatisierte Tests. Ein Szenario stellt eine spezifische Verhaltensweise unter bestimmten Bedingungen dar. Szenarien basieren typischerweise auf den Abnahmekriterien, Beispielen und Anwendungsfällen (Use-Cases) der User-Story. BDD-Szenarien sollten in einer Sprache geschrieben werden, die von allen Teammitgliedern, unabhängig vom technischen Know-how, verstanden werden kann.

Daher wird die Verwendung von natürlicher Sprache wie z. B. Englisch zur Formulierung von Szenarien stark bevorzugt. Darüber hinaus etablieren Teams, die BDD anwenden, eine ubiquitäre (allgemein verwendete, einheitliche) Sprache (siehe [Evans03]), die im Wesentlichen ein für alle Teammitglieder klares und eindeutiges und überall verwendetes Fachvokabular festlegt.

BDD-Szenarien bestehen typischerweise aus drei Abschnitten (siehe [Gherkin]):

1. **Gegeben** (Given) – beschreibt den Zustand der Umgebung (Vorbedingungen), bevor das Verhalten ausgelöst wird.
2. **Wenn** (When) – beschreibt die Aktionen, die das Verhalten auslösen.
3. **Dann** (Then) – beschreibt die erwarteten Ergebnisse des Verhaltens.

Das Extrahieren von Szenarien aus User-Storys beinhaltet:

- Identifizieren aller durch eine User-Story spezifizierten Abnahmekriterien und Schreiben von Szenarien für jedes Abnahmekriterium. Einige Abnahmekriterien können mehrere Szenarien erfordern.
- Identifizieren von funktionalen Anwendungsfällen (Use-Cases) und Beispielen sowie Schreiben von Szenarien für jeden Anwendungsfall und jedes Beispiel.

Viele Anwendungsfälle und Beispiele sind an Bedingungen geknüpft. Daher ist es entscheidend, jede der Bedingungen zu identifizieren.

- Erstellen von Szenarien während des explorativen Testens, mit deren Hilfe zugehörige vorhandene Verhaltensweisen, potentiell widersprüchliche Verhaltensweisen, minimale Zustände, alternative Abläufe usw. identifiziert werden können.
- Suchen nach wiederholter Verwendung von Schritten oder Gruppen von Schritten, um sich wiederholende Arbeit zu vermeiden.
- Identifizieren von Bereichen, die Zufallsdaten oder synthetische Daten benötigen.
- Identifizieren von Schritten, die Mocks, Stubs oder Treiber benötigen, um die Abgeschlossenheit aufrechtzuerhalten und eine integrative Ausführung oder möglicherweise aufwendige Prozesse zu vermeiden.
- Sicherstellen, dass die Szenarien atomar sind und nicht gegenseitig ihren Status bzw. Zustand beeinflussen (abgeschlossen).
- Entscheiden, ob die “Wenn“-Abschnitte auf einen Schritt oder Aktion beschränkt werden sollen nach dem Prinzip jeder Test prüft nur einen Aspekt oder ob sie aus anderen Überlegungen heraus optimiert werden sollen, wie z. B. die Testdurchführungsgeschwindigkeit.

Empfohlene Richtlinien für die Formulierung von Szenarien sind unter anderem:

- Das Szenario sollte eine spezifische Verhaltensweise beschreiben, welche das System aus der Sichtweise eines spezifischen Benutzers unterstützt.
- Das Szenario sollte die dritte Person bei der Beschreibung der Abschnitte (“Gegeben“, “Wenn“, “Dann“) verwenden, um den Zustand und die Interaktionen aus der Sichtweise des Benutzers zu beschreiben.
- Die Szenarien sollten abgeschlossen und atomar sein, damit sie in beliebiger Reihenfolge ablaufen können und sich nicht gegenseitig beeinflussen oder voneinander abhängen. Die “Gegeben“-Abschnitte sollten das System in den erforderlichen Zustand versetzen, damit die “Wenn“-Abschnitte wie erwartet ausgeführt werden können.
- Die “Wenn“-Abschnitte sollten eher die semantischen als die spezifischen technischen Aktionen beschreiben, die ein Benutzer ausführt, es sei denn, es besteht eine besondere Notwendigkeit, eine spezifische Aktion zu testen. Zum Beispiel ist "Der Benutzer bestätigt die Bestellung" (eine semantische Aktion) im Allgemeinen besser als "Der Benutzer klickt auf den Bestätigungsbutton" (eine technische Aktion), es sei denn, der Button selbst muss getestet werden.
- Die “Dann“-Abschnitte sollten konkrete Beobachtungen oder Zustände beschreiben. Sie sollten keine generischen Erfolgs- oder Fehlerzustände angeben.

2.1.3 Abnahmetestgetriebene Entwicklung (Acceptance Test-Driven Development, ATDD)

Abnahmetestgetriebene Entwicklung (ATDD) unterstützt die Softwareprojekte dabei, bereitzustellende Ergebnisse an den Kundenbedürfnissen auszurichten. Abnahmetests sind Spezifikationen für das gewünschte Verhalten und die Funktionalität eines Systems. Eine User-Story repräsentiert ein Teil Funktionalität, die dem Kunden einen Mehrwert bietet. Abnahmetests verifizieren, dass diese Funktionalität korrekt realisiert ist. Die User-Story wird von den Entwicklern in eine Reihe von Aufgaben zerlegt, die zur Realisierung dieser Funktionalität erforderlich sind.

Entwickler können diese Aufgaben durch die Anwendung von TDD umsetzen. Wenn eine bestimmte Aufgabe abgeschlossen ist, gehen die Entwickler zur nächsten Aufgabe über, bis die User-Story abgeschlossen ist, was durch erfolgreich durchgeführte Abnahmetests erkennbar wird. Sowohl BDD als auch ATDD sind kundenorientiert, während TDD entwicklerorientiert ist. BDD und ATTD ähneln sich insofern, dass sie beide das gleiche Ergebnis liefern: ein gemeinsames Verständnis dessen, was realisiert werden soll, und wie man es korrekt realisiert. BDD ist eine strukturierte Methode zum Schreiben von Testfällen (z. B. Abnahmetests) unter Verwendung der zuvor besprochenen “Gegeben“, “Wenn“, “Dann“-Syntax.

ATDD trennt die Testabsicht, die in einem für den Menschen lesbaren Format wie z. B. Klartext ausgedrückt wird, von der Testrealisierung, die vorzugsweise automatisiert ist. Diese wichtige Trennung bedeutet, dass fachlich-orientierte Teammitglieder wie z. B. Product Owner, Analysten und Tester bei der Beschreibung von testbaren Beispielen (oder Abnahmetests) eine aktive Rolle übernehmen können um die Entwicklung zu lenken. Stakeholder mit unterschiedlichen Rollen und Sichtweisen, wie z. B. Kunde, Entwickler und Tester kollaborieren und diskutieren gemeinsam eine potentielle User-Story (oder eine andere Form der Anforderung) um ein gemeinsames Verständnis des zu lösenden Problems zu erreichen, sich gegenseitig offene Fragen zur Funktionalität zu stellen und konkrete, testbare Beispiele für ein gefordertes Verhalten zu sondieren.

Die vereinbarten Beispiele (und die Diskussionen, die zu diesen Beispielen führen) sind an sich schon wertvolle Ergebnisse. Es ist wichtig zu bedenken, dass sie zwar auch als Abnahmetests automatisiert werden können, dies jedoch nicht immer notwendig oder kosteneffizient ist. Diese Betonung der Idee von wertvollen Beispielen anstelle von Tests ist beabsichtigt und kann ein wichtiger Kunstgriff sein, um alle Mitglieder eines Teams zu ermutigen, sich am Entdeckungsprozess zu beteiligen.

Dies zeigt, dass der agile Tester in dieser Situation eine wichtige Rolle spielt. Während der Diskussionen denkt der Tester möglicherweise in Form von Testverfahren, wie z. B. Äquivalenzklassenbildung und Grenzwertanalyse. Agile Tester können diesen analytischen Ansatz nutzen, um Fragen an den Product Owner zu stellen wo beispielsweise interessante Verhaltensweisen und/oder Grenzfälle aufzufinden sind. Die Absicht sollte immer sein, die ganze Gruppe zu ermutigen, die Schlüsselbeispiele

zu betrachten und zu finden. Eine interessante Kombination von Eingaben aufzuwerfen und die Gruppe zu fragen, ob sie sich über die erwarteten Ausgaben und Ergebnisse einig sind, ist eine gute Möglichkeit, potentielle Bereiche der Unsicherheit oder der unvollständigen Analyse zu erkunden.

Die Spezifikation durch Beispiele (Specification by Example, SBE) bezieht sich auf eine Sammlung nützlicher Muster (pattern), die es einem agilen Team ermöglichen, die von den Stakeholdern aus den Geschäftsbereichen geforderten maßgeblichen Ergebnisse und die zur Erreichung dieser Ergebnisse erforderlichen Software-Verhaltensweisen zu identifizieren, zu diskutieren und zu bestätigen. Der Begriff wurde häufig verwendet, um die Bedeutung von abnahmetestgetriebener Entwicklung (ATDD) einzubeziehen und zu erweitern.

2.2 Erfahrungsbasiertes Testen im agilen Umfeld

Die verschiedenen Charakteristika agiler Projekte wie der spezifische agile Ansatz, die Iterationsdauer, die eingesetzten Teststufen, die Risikostufe von Projekt und Produkt, die Qualität der Anforderungen, der Kompetenz- und Erfahrungsstand der Teammitglieder, die Projektorganisation etc. können die Balance zwischen automatisierten Tests, explorativen Tests und manuellen Black-Box-Tests in einem agilen Projekt beeinflussen.

2.2.1 Erfahrungsbasierte Testverfahren und Black-Box-Tests kombinieren

Im Rahmen einer Risikoanalyse werden die Risikostufen (z. B. hoch, mittel, niedrig) für die einzelnen Systemfeatures und Funktionalitäten ermittelt. Der nächste Schritt ist die richtige Mischung und Balance aus automatisierten Tests, explorativen Tests und manuellen Black-Box-Tests für eine bestimmte Risikostufe herauszuarbeiten. Die folgende Tabelle veranschaulicht diese Idee. In dieser Tabelle sind die Risikostufen vertikal und die drei Testvorgehensweisen horizontal aufgeführt. Die folgenden Symbole werden zur Beschreibung verwendet:

++ (sehr empfohlen)

+ (empfohlen)

o (neutral)

- (nicht empfohlen)

- - (nicht zu verwenden)

Die folgende Tabelle zeigt beispielhaft eine Mischung aus verschiedenen Testverfahren (explorativ, automatisiert und manuell), die zum Test eines sicherheitskritischen Systems angewendet werden kann. Für andere spezifische Projektcharakteristika muss diese Tabelle angepasst werden.

RISIKOSTUFE	AUTOMATISIERTE TESTS	EXPLORATIVE TESTS	BLACK-BOX-TESTS
HOCH	++	+	++
MITTEL	+	+	+
NIEDRIG	o	++	+

Tabelle 1: Beispiel für einen Testverfahren-Mix zum Test eines unternehmenskritischen und/oder sicherheitskritischen Systems

Aus der ersten Zeile in Tabelle 1 wird ersichtlich, dass in dieser Situation eine Kombination von automatisierten Tests und Black-Box-Tests zusätzlich zu einem explorativen Testansatz sehr zu empfehlen wäre. Die Entscheidung für oder gegen eine Automatisierung wird aber auch von vielen anderen Faktoren beeinflusst.

Nachfolgend ist ein Beispiel für eine Mischung verschiedener Testverfahren bei der Anwendung zum Test eines nicht sicherheitskritischen Systems dargestellt.

RISIKOSTUFE	AUTOMATISIERTE TESTS	EXPLORATIVE TESTS	BLACK-BOX-TESTS
HOCH	+	++	+
MITTEL	o	++	o
NIEDRIG	- -	++	- -

Tabelle 2: Beispiel für einen Testverfahren-Mix zum Test eines nicht unternehmenskritischen und/oder nicht sicherheitskritischen Systems

Aus der letzten Zeile in Tabelle 2 wird ersichtlich, dass in dieser Situation ein explorativer Testansatz sehr zu empfehlen ist, während andere Ansätze möglicherweise nicht verwendet werden. In jeder Situation (sicherheitskritisch oder nicht) hängt die spezifische Mischung von den jeweils vorliegenden Projektcharakteristika ab.

2.2.2 Erstellung von Test-Chartas und Interpretation der Ergebnisse

Bevor eine geeignete Test-Charta erstellt werden kann, sollten zunächst die vorhandenen Epics und User-Storys ausgewertet werden (siehe Kapitel 1).

Bei der Analyse der Epics und User-Storys zur Erstellung der Test-Charta sollte folgendes berücksichtigt werden:

- Wer sind die Benutzer (User), die in dieser Epic oder User-Story vorkommen?
- Was ist die Hauptfunktionalität des Epics oder der User-Story?
- Welche Aktionen kann ein Benutzer ausführen? (Dies kann u. U. der Liste der Abnahmekriterien entnommen werden, die für eine User-Story definiert ist.)

- Wird das Ziel der User-Story durch Fertigstellung des Features oder der Funktionalität erreicht? (Oder gibt es weitere Testaufgaben, die sich auf die Definition of Done auswirken?)

Die Granularität einer Test-Charta ist wichtig. Die Test-Charta sollte nicht zu begrenzt sein, da sie einen Bereich um ein identifiziertes Problem (reaktiv, regressiv) oder einen Bereich um eine User-Story oder eine Epic (proaktiv, aufdeckend) erkunden sollte.

Die Test-Charta sollte auch nicht zu feingranular sein, da sie in eine Timebox von 60 bis 120 Minuten passen sollte. Das Ziel einer explorativen Testsitzung ist es dabei zu unterstützen, eine gut begründete Qualitätsentscheidung in Bezug auf ein bestimmtes Phänomen, einen fehlerbehafteten Bereich, usw. zu treffen. Die Ergebnisse einer explorativen Testsitzung sollten genügend Informationen liefern, um eine solche Entscheidung zu treffen.

Test-Chartas können unter Verwendung von Flipcharts, Tabellenkalkulationen, Dokumenten, einem bestehenden Testmanagementsystem, Personas, Mind-Mapping und der Einbindung des gesamten Teams erstellt werden. Explorative Tester verwenden Heuristiken, um ihre Kreativität beim Schreiben und Durchführen von explorativen Testsitzungen zu fördern. Diese können auch zur Erstellung von Test-Chartas und zum kreativen Denken bei der Analyse von User-Storys und Epics herangezogen werden. Beispiele für Heuristiken siehe [Whittaker09] und [Hendrickson13].

Alle bei den explorativen Tests gewonnenen Erkenntnisse sollten dokumentiert werden. Die Ergebnisse des explorativen Testens sollten Einsichten für ein besseres Testdesign, Ideen für das Testen des Produkts und Ideen für weitere Verbesserungen liefern. Zu den Erkenntnissen, die beim explorativen Testen dokumentiert werden sollten, gehören beispielsweise Fehlerzustände, Ideen, Fragen und Verbesserungsvorschläge.

Für die Dokumentation explorativer Testsitzungen können Werkzeuge eingesetzt werden. Dazu gehören z. B. Videoaufnahme-, Protokollierungs- und Planungswerkzeuge. Die Dokumentation sollte das erwartete Ergebnis enthalten. In einigen Fällen sind Stift und Papier ausreichend, je nach Menge der zu sammelnden Informationen.

Bei der Zusammenfassung einer explorativen Testsitzung werden die Informationen während der Nachbesprechung gesammelt und zusammengefasst, um den Status des Fortschritts, der Überdeckung und der Effizienz der Testsitzung darzustellen. Diese zusammenfassenden Informationen können als Managementbericht oder in Retrospektiven auf jeder Ebene und in jeder Skalierung (ein einzelnes Team, mehrere Teams oder agile Implementierung im Großen / ‚large scale') verwendet werden. Es kann jedoch eine ziemliche Herausforderung sein, geeignete Testmetriken in Verbindung mit explorativen Testsitzungen zu bestimmen.

2.3 Aspekt der Codequalität

Die Steuerung der technischen Schulden, insbesondere im Hinblick auf die Aufrechterhaltung einer hohen Codequalität während des gesamten Releases, ist in agilen Projekten sehr wichtig. Um dieses Ziel zu erreichen, werden verschiedene Verfahren eingesetzt.

2.3.1 Refactoring

Refactoring ist eine Möglichkeit, Code auf effiziente und kontrollierte Weise zu bereinigen, indem das Design von bestehendem Code und Testfällen klarer und einfacher gestaltet wird, ohne das Verhalten zu verändern. In agilen Projekten sind die Iterationen kurz, wodurch eine kurze Feedback-Schleife für alle Teammitglieder entsteht. Kurze Iterationen sind auch eine Herausforderung für Tester beim Versuch, eine angemessene Überdeckung zu erreichen. Aufgrund der Natur von Iterationen und der Tatsache, dass dabei die Funktionalität wächst und Features im Laufe der Zeit hinzugefügt und erweitert werden, müssen Testfälle, die für ein Feature in einer früheren Iteration erstellt wurden, in späteren Iterationen oft angepasst oder sogar komplett neu entworfen werden. Durch Verwendung einer evolutionären Vorgehensweise für den Testentwurf kann die Aktualisierung und das Refactoring der Tests die Änderungen von Features kompensieren sowie sicherstellen, dass die Tests mit der Funktionalität des Produkts im Einklang bleiben.

Nachdem die User-Storys verstanden und Abnahmekriterien für jede von ihnen geschrieben wurden, können die Auswirkungen der Funktionalität der aktuellen Iteration auf die vorhandenen Regressionstests (manuell und automatisiert) analysiert werden, wobei ein Refactoring und/oder eine Erweiterung der Tests erforderlich sein kann. Die Teams pflegen und erweitern den Code von Iteration zu Iteration in erheblichem Umfang. Ohne kontinuierliches Refactoring ist dies nur schwer zu bewerkstelligen.

Das Refactoring der Testfälle kann wie folgt durchgeführt werden:

- **Identifizieren:** Vorhandene Tests, die ein Refactoring benötigen durch ein Review oder eine Ursachenanalyse identifizieren.
- **Analysieren:** Auswirkungen der geänderten Tests auf die gesamte Menge der Regressionstests analysieren.
- **Refactoring:** Änderungen an der internen Struktur der Tests vornehmen, um die Verständlichkeit zu erhöhen und den Aufwand für Änderungen zu reduzieren, ohne dabei ihr beobachtbares Verhalten zu verändern.
- **Wiederholen:** Tests wiederholen, Ergebnisse überprüfen und gegebenenfalls gefundene Fehlerzustände dokumentieren. Das Refactoring sollte das Ergebnis der Testdurchführung nicht beeinflusst haben.

- **Auswerten:** Ergebnisse der wiederholten Testdurchführungen überprüfen und diese Phase beenden, sobald die Tests eine vom Team definierte und akzeptierte Mindestqualität erreicht haben.

2.3.2 Code-Reviews und statische Codeanalyse zur Identifikation von Fehlerzuständen und technischen Schulden

Ein Code-Review ist eine systematische Untersuchung des Codes durch zwei oder mehr Personen (von denen eine in der Regel der Autor ist). Statische Codeanalyse ist die systematische Untersuchung von Code durch ein Werkzeug. Beide sind effektive und weitverbreitete Methoden zur Aufdeckung und Identifizierung von Problemen, die die Codequalität beeinträchtigen. Code-Reviews und statische Codeanalysen liefern konstruktives Feedback, das bei der Identifizierung von Fehlerzuständen und der Bewältigung technischer Schulden hilft.

Hindernisse wie Ressourcenbegrenzung, höhere technische Komplexität als erwartet, schnell wechselnde Prioritäten und technische Limitierungen können die Bemühungen der Entwickler behindern, Code von guter Qualität zu schreiben, und sie daher zu Kompromissen zwingen, die die Qualität des Codes zugunsten sofortiger Ergebnisse verringern. Diese Kompromisse können Fehlerzustände zur Folge haben und technische Schulden verursachen.

Technische Schulden bezieht sich auf den erhöhten Aufwand, der in der Zukunft für die Realisierung einer besseren Lösung erforderlich ist (einschließlich der Beseitigung verdeckter Fehlerzustände), als Folge, wenn man sich heute für eine minderwertige, aber leichter zu realisierende Lösung entscheidet. Technische Schulden entstehen oft unbeabsichtigt, durch kaum wahrnehmbare Kompromisse oder eine allmähliche Anhäufung kleiner oder unbemerkter Änderungen während der Weiterentwicklung der Software. Code-Reviews und statische Codeanalyse helfen dabei verschiedene Ursachen für technische Schulden zu identifizieren wie z. B. erhöhte Komplexität, zirkuläre Bezüge, Konflikte zwischen verschiedenen Codemodulen, schlechte Codeüberdeckung und nicht sicherer Code. Andere Arten von technischen Schulden können auch bei z. B. Testartefakten, Infrastruktur und in der Continuous Integration (CI) Pipeline auftreten.

Wenn technische Schulden bewusst eingegangen (als Folge anderer Entscheidungen oder als Kompromiss) oder durch Code-Reviews oder statische Codeanalyse aufgedeckt werden, sollten Anstrengungen unternommen werden, die technischen Schulden zu reduzieren. Dabei sollte man sich möglichst sofort dem Problem stellen. Wenn das nicht möglich ist, sollten Aufgaben zur Beseitigung der technischen Schulden ins (Product-) Backlog aufgenommen werden.

Der Zielkonflikt zwischen dem zusätzlichen Zeitaufwand für Codeanalysen und Code-Reviews einerseits und dem Eingehen technischer Schulden andererseits, geht fast immer zugunsten von Analysen und Reviews des Codes aus. Wenn sich mit Fehlerzuständen und technischen Schulden behafteter Code ausbreitet, wird es immer

schwieriger, kostspieliger und zeitaufwändiger, diesen zu korrigieren, ohne dabei andere Teile des Systems zu beschädigen. Codeanalyse und Reviews können die Qualität des Codes verbessern und den Zeitaufwand insgesamt reduzieren.

Neben der Unterstützung bei der Identifizierung von Fehlerzuständen und der Verwaltung der technischen Schulden bieten Codeanalysen und Reviews zusätzliche Vorteile:

- Schulung und Wissensaustausch
- Verbesserung der Robustheit, Wartbarkeit und Lesbarkeit des Codes
- Überwachung und Pflege einheitlicher Programmierstandards

Code-Reviews

Durch die Teilnahme an Code-Reviews können Tester ihre besondere Sichtweise nutzen, um wertvolle Beiträge zur Codequalität zu leisten, indem sie gemeinsam mit den Entwicklern mögliche Fehlerzustände erkennen und technische Schulden bereits in einem sehr frühen Stadium vermeiden. Tester sollten in der Lage sein, die im geprüften Code verwendete Programmiersprache lesen zu können. Die Tester müssen jedoch keine vertieften Programmierkenntnisse haben, um effektiv an Code-Reviews teilnehmen zu können. Sie können ihre Expertise auf vielfältige Weise einbringen, z. B. indem sie Fragen zum Code-Verhalten stellen, Anwendungsfälle (Use-Cases) vorschlagen, die möglicherweise nicht berücksichtigt wurden oder Code-Metriken überwachen, die auf Qualitätsprobleme hinweisen könnten. Code-Reviews bieten auch die Möglichkeit, Wissen zwischen Entwicklern und Testern auszutauschen. Zwei der größten Herausforderungen für Code-Reviews in agilen Projekten sind kurze Iterationen und die zu ihrer Durchführung benötigte Zeit. Es ist wichtig, Code-Reviews einzuplanen und die dafür benötigte Zeit in jeder Iteration freizuhalten.

Code-Reviews sind manuelle Tätigkeiten, eventuell unterstützt durch Tools, die von oder mit anderen Personen (zusätzlich zum Autor) durchgeführt werden. Normalerweise werden Teamleiter oder erfahrenere Entwickler im Team das Code-Review durchführen, obwohl es auch mit anderen Teammitgliedern durchgeführt werden kann. Für Tester und andere Nicht-Entwickler ist es oft von Vorteil, an Code-Reviews teilzunehmen.

Unterschiedliche Ansätze für Code-Reviews unterscheiden sich in ihrem Grad an Formalität und Stringenz (siehe [Wiegers02]). Formalere und rigorosere Ansätze sind in der Regel gründlicher, aber auch zeitaufwändiger. Weniger formale und rigorose Ansätze sind etwas weniger gründlich, können jedoch in wesentlich kürzerer Zeit durchgeführt werden. Es gibt verschiedene Arten von Peer-Reviews, wobei agile Teams eher schnellere Reviews bevorzugen, die regelmäßig durchgeführt werden, üblicherweise vor jeder Integration.

Code-Reviews können durch den Gutachter (auch Reviewer genannt) und dem Autor/Entwickler in einer gemeinsamen Sitzung durchgeführt werden. Dieser Modus, der bei Ad-hoc-Reviews und bei der Programmierung in Paaren (Pair Programming) üblich ist, ermöglicht eine ausgezeichnete Kommunikation und fördert gründlichere Analysen sowie einen besseren Wissensaustausch. Er kann auch zum Teamzusammenhalt und zur Moral des Teams beitragen.

Bei verteilten Teams oder Teams, die einen eher etwas isolierteren Ansatz bevorzugen, wird der Code-Review-Prozess durch ein Konfigurationsmanagementsystem unterstützt. Der Prozess ist in der Regel teilautomatisiert als Teil des Continuous Integration Process. Dieser Prozess kann Code-Reviews von einzelnen Gutachtern pro Review-Runde oder gemeinschaftliche Team-Reviews unterstützen.

Statische Codeanalyse

Bei der statischen Codeanalyse analysiert ein Werkzeug den Code und sucht nach bestimmten Aspekten, ohne den Code auszuführen. Die Ergebnisse der statischen Codeanalyse können auf eindeutige Probleme im Code hinweisen oder indirekte Indikatoren liefern, die dann eine weitere Untersuchung erforderlich machen.

Viele Entwicklungswerkzeuge, insbesondere integrierte Entwicklungsumgebungen (IDE), können schon während des Schreibens des Codes eine statische Codeanalyse durchführen. Dies bietet den Vorteil eines sofortigen Feedbacks, wobei es aber unter Umständen sein kann, dass nur ein Teil der Analyse zur Anwendung kommt, die während der Continuous Integration durchgeführt werden.

3 Testautomatisierung – 135 Minuten

Schlüsselbegriffe

Datengetriebenes Testen, schlüsselwortgetriebenes Testen, Testablauf, Testvorgehensweise

Lernziele für Testautomatisierung

ATT-3.x (K1) Schlüsselbegriffe

3.1 Testautomatisierungsverfahren

ATT-3.1.1 (K3) Datengetriebenes und schlüsselwortgetriebenes Testen zur Entwicklung automatisierter Testskripte anwenden können

ATT-3.1.2 (K2) Verstehen können, wie Testautomatisierung bei einer bestimmten Testvorgehensweise in einem agilen Umfeld angewendet wird

ATT-3.1.3-1 (K2) Testautomatisierung verstehen können

ATT-3.1.3-2 (K2) Unterschiede zwischen verschiedenen Testvorgehensweisen verstehen können

3.2 Automatisierungsgrad

ATT-3.2.1-1 (K2) Faktoren verstehen können, die bei der Bestimmung des Testautomatisierungsgrads zu berücksichtigen sind, um mit der Geschwindigkeit der Bereitstellung mitzuhalten

ATT-3.2.1-2 (K2) Herausforderungen der Testautomatisierung im agilen Umfeld verstehen können

3.1 Testautomatisierungsverfahren

3.1.1 Datengetriebenes Testen

Motivation

Datengetriebenes Testen ist ein Testautomatisierungsverfahren, das den Aufwand für die Entwicklung und Wartung von Testfällen mit identischen Testschritten, aber unterschiedlichen Kombinationen von Testdateneingaben minimiert. Datengetriebenes Testen ist ein etabliertes Verfahren, das nicht spezifisch für das Testen in agilen Projekten ist. Aufgrund seiner Fähigkeit, den Entwicklungs- und Wartungsaufwand für die Testautomatisierung zu senken, sollte dieses Verfahren in jedem agilen Projekt als Teil der Testautomatisierungsstrategie berücksichtigt werden.

Konzept

Die Grundidee des datengetriebenen Testens ist die Trennung der Testlogik von den Testdaten. Stattdessen werden im Testablauf Testdatenvariablen benannt, die auf Testdatenwerte verweisen, welche in einer separaten Testdatenliste/-tabelle bereitgestellt werden. Der Testablauf kann dann mit verschiedenen Testdatensätzen wiederholt ausgeführt werden. Weitere Details und Beispiele siehe [AdvancedTestAutomationEngineer].

Vorteile für agile Teams

- Agile Teams können sich von Iteration zu Iteration schnell an die sich ändernden/erweiternden Funktionalitäten eines Produkts anpassen, da das Ändern/Hinzufügen neuer Datenkombinationen einfach ist und wenig oder keine Auswirkungen auf die bestehende Testautomatisierung hat.
- Agile Teams können die erforderliche Testüberdeckung einfach nach oben oder unten anpassen, indem sie Einträge in der Testdatentabelle hinzufügen, ändern oder entfernen. Auf diese Weise können agile Teams auch die Testdurchführungszeiten steuern, um die Bedingungen für ein Continuous Deployment (kontinuierliche Bereitstellung/Inbetriebnahme) zu erfüllen.
- Datengetriebenes Testen unterstützt die Philosophie des multidisziplinären Arbeitens (working cross-functionally), da Testdatentabellen einfacher zu verstehen sind als Testskripte und daher eine effektivere Teilnahme von Teammitgliedern mit weniger technischem Know-how ermöglichen.
- Da Testdatentabellen leichter verständlich sind, unterstützt datengetriebenes Testen ein frühes Feedback zu Testfällen und Abnahmekriterien von Teammitgliedern mit weniger/keinem technischen Know-how und Kunden/Anwendern.
- Datengetriebenes Testen hilft agilen Teams Testautomatisierungsaufgaben effizienter zu erledigen, da es nicht nur den Aufwand für die Entwicklung neuer Tests verringert, sondern auch die Wartung vorhandener datengetriebener Tests reduziert.

Einschränkungen für agile Teams

Es gibt zwar keine spezifischen Einschränkungen für die Anwendung datengetriebenen Testens in einem agilen Kontext, aber agile Teams sollten sich der grundsätzlichen Einschränkungen bei der Anwendung dieses Verfahrens bewusst sein. Weitere Details hierzu siehe [AdvancedTestAutomationEngineer].

Werkzeuge

- Das Bearbeiten, Speichern und Verwalten der Testdaten erfolgt in der Regel mit Hilfe von Tabellenblättern oder Textdateien.

- Die meisten Testautomatisierungswerkzeuge/-sprachen bieten integrierte Funktionsaufrufe zum Lesen von Testdaten aus Tabellenblättern oder Textdateien.

3.1.2 Schlüsselwortgetriebenes Testen

Motivation

Ein Nachteil der Testautomatisierung ist, dass automatisierte Testskripte viel schwieriger zu verstehen sind als in natürlicher Sprache beschriebene Testabläufe für den manuellen Test. Die Anwendung einer schlüsselwortgetriebenen Testautomatisierung trägt zur Verbesserung der Lesbarkeit, Verständlichkeit und Wartbarkeit automatisierter Testskripte bei.

Konzept

Die Grundidee des schlüsselwortgetriebenen Testens besteht darin, eine Reihe von Schlüsselwörtern zu definieren, die den Anwendungsfällen (Use-Cases) eines Produkts und/oder des Geschäftsbereiches des Kunden entnommen wurden, und diese als Vokabular zur Formulierung von Testabläufen zu verwenden. Die daraus resultierenden automatisierten Testskripte sind aufgrund ihrer semi-natürlichen Sprache einfacher zu verstehen als der Quellcode in Programmier- bzw. Skriptsprache. Weitere Details und Beispiele siehe [AdvancedTestAutomationEngineer].

Es gibt verschiedene Arten, wie ein Testablauf mit Hilfe von Schlüsselwörtern beschrieben werden kann (siehe [Linz 14], Kapitel 6.4.2 Schlüsselwortgetriebenes Testen):

- In Listenform: Ein Testablauf ist demnach eine einfache Abfolge/Auflistung von Schlüsselwörtern
- In Form von BDD: Ein Testablauf (oder ein Szenario) wird als Satz in semi-natürlicher Sprache formuliert, wobei die Schlüsselwörter als „ausführbare“ Teile des Satzes verwendet werden (siehe Kapitel 2.1.2 (BDD))
- In Form einer domänenspezifischen Sprache (Domain-Specific Language, DSL): Regeln, welche auf dem Konzept der domänenspezifischen Sprache basieren (siehe [Fowler/Parsons10]), definieren, wie die Schlüsselwörter (und möglicherweise zusätzliche Sprachelemente) kombiniert werden können.

Vorteile für agile Teams

- Durch die Definition von Schlüsselwörtern oder einer domänenspezifischen Sprache (DSL) erstellt und standardisiert ein agiles Team sein eigenes domänenbezogenes Teamvokabular, mit dessen Hilfe die Teammitglieder klarer und präziser kommunizieren und Missverständnisse vermeiden können.

- Agile Teams können schnell qualifizierteres Kunden-/Benutzer-Feedback sammeln. Testabläufe, die aus Schlüsselwörtern zusammengestellt und/oder in Form von BDD/DSL geschrieben sind, können von den Kunden besser verstanden werden als ein Testskript in Programmiersprache. Informelle Abnahmekriterien können formalisiert werden, ohne die „natürlichsprachliche Lesbarkeit“ zu verlieren. Dies kann helfen, die Geschäftslogik selbst und damit die Interpretation der Abnahmekriterien besser zu verstehen.
- Die Art und Weise, wie die Testfälle erstellt werden, weist auf die Erstellung und Verwaltung einer lebenden Dokumentation hin.
- Schlüsselwortgetriebenes Testen hilft agilen Teams, ihre Testautomatisierungsaufgaben multidisziplinär zu erfüllen, indem die Teammitglieder mit weniger technischem Know-how in die Testautomatisierung einbezogen werden, da die Zusammenstellung von Testabläufen aus vorhandenen Schlüsselwörtern keine Programmierkenntnisse erfordert.
- Die Änderung des Verhaltens eines definierten Schlüsselwortes erfordert wesentlich weniger Aufwand als die Änderung desselben Verhaltens über mehrere Testabläufe hinweg. Dies kann den Wartungsaufwand erheblich reduzieren und setzt Ressourcen frei, die für die Realisierung neuer Tests verwendet werden können.
- Ein agiles Team kann dieses Verfahren (und damit seine Vorteile) über die gesamte Testpyramide hinweg anwenden, da Schlüsselwörter implementiert werden können, um das Testobjekt über beliebige Schnittstellen anzusteuern; von Tests auf Benutzungsschnittstellenebene bis hinunter zur API-Ebene (z. B. über API-Aufrufe, REST-Aufrufe, Soap-Aufrufe). Damit ist das Konzept auch auf Unit- und Integrationstests anwendbar und nicht nur auf Systemtests über die Benutzungsschnittstelle beschränkt, wie oft angenommen wird. Es kann jedoch sein, dass dadurch eine unnötige Abstraktionsebene bei Unittests hinzukommt.

Einschränkungen für agile Teams

Neben den grundsätzlichen Einschränkungen dieses Ansatzes, wie in [AdvancedTestAutomationEngineer] beschrieben, sollten sich agile Teams auch der folgenden Einschränkungen und potenziellen Fallstricke bewusst sein:

- Zur Ausführung von schlüsselwortgetriebenen Testabläufen wird ein geeignetes Ausführungsframework (z. B. mit einem Schlüsselwortinterpreter) benötigt. Es ist nicht empfehlenswert, das Framework von Grund auf neu zu erfinden. Das Team erreicht eine höhere Geschwindigkeit, wenn ein vorhandenes Framework oder Werkzeug verwendet wird, welches schlüsselwortgetriebenes Testen unterstützt. Dies ist insbesondere dann der Fall, wenn das Team einem BDD- oder DSL-Ansatz folgt.
- Ein neues Schlüsselwort stabil und wartbar zu implementieren ist schwierig und erfordert Erfahrung und gute Programmierkenntnisse. Die Aufgaben der

Implementierung von Schlüsselwörtern konkurrieren auch mit den Aufgaben der Programmierung des Produktes. Die resultierende Geschwindigkeit der Testautomatisierung könnte daher geringer sein als erwartet.

- Die Schlüsselwörter (Benennung, Abstraktionsebene) und/oder die DSL (Grammatikregeln) müssen gut entworfen sein. Andernfalls werden die Verständlichkeit und/oder die Skalierbarkeit nicht den Erwartungen entsprechen.
- Die Schlüsselwörter müssen auch angemessen verwaltet werden. Wenn dies nicht geschieht, wird sich die Implementierung von Schlüsselwörtern nicht auszahlen, da mehrere Implementierungen von Synonymen auftreten können oder Schlüsselwörter werden nicht oder nur selten in Testfällen verwendet. Um diese Fallstricke zu vermeiden, sollte das Team ein Teammitglied für die Verwaltung des Schlüsselwörtervokabulars verantwortlich machen.
- Das Team sollte sich bewusst sein, dass die Anwendung von schlüsselwortgetriebenem Testen einige Vorabinvestitionen erfordert (z. B. für die Definition der Schlüsselwörter/Domänensprache, die Auswahl eines geeigneten Frameworks, die Implementierung der ersten Reihe von Schlüsselwörtern) und zu Beginn eines Projekts die Geschwindigkeit der Testautomatisierung dadurch möglicherweise reduziert wird.

Werkzeuge

- Wie beim datengetriebenen Testen ist die Verwendung von Tabellenblättern, Textdateien oder einfachen Dateien ein üblicher aber begrenzter Ansatz zur Bearbeitung, Speicherung und Verwaltung von schlüsselwortgetriebenen Tests.
- Mehrere Testframeworks, Testdurchführungs- und Testmanagement-Werkzeuge bieten eine integrierte Unterstützung für schlüsselwortgetriebenes Testen (je nach Werkzeuganbieter oder Framework als "schlüsselwortgetriebenes Testen", "interaktionsbasiertes Testen", "Geschäftsprozesstest" oder "verhaltensgetriebenes Testen" bezeichnet). Verfügbare Werkzeuge siehe [ToolList].

3.1.3 Anwenden der Testautomatisierung auf eine gegebene Testvorgehensweise

Testautomatisierung ist kein Testziel. Testautomatisierung ist eine Strategie, die, wenn sie angemessen verfolgt wird, umfangreichere strategische Testziele fördern kann, indem sie die Testeffizienz erhöht, das Testen gegen bestimmte Fehlertypen (z. B. Performanz- und Zuverlässigkeitsfehlerzustände) effektiver macht oder eine frühere Entdeckung von Fehlerzuständen ermöglicht. Die Strategie passt sich dem Kontext an und entwickelt sich somit kontinuierlich weiter.

Die Testautomatisierung nimmt oft viele verschiedene Formen an und beinhaltet viele verschiedene Werkzeuge, je nach Kontext und Bedarf des Teams. In großen Projekten gibt es in der Regel nicht nur eine Lösung, die allen Anforderungen gerecht wird,

sodass mehrere Testautomatisierungsstrategien zur Anwendung kommen. Die Anwendung der Testautomatisierung muss der Teststrategie der Organisation und der Testvorgehensweise des jeweiligen Projekts angemessen sein.

Testautomatisierung ist mehr als nur die Automatisierung der Testdurchführung. Testautomatisierung kann eine wichtige Rolle bei der Konfiguration der Testumgebung, dem Release Management der Testartefakte, dem Testdatenmanagement und dem Vergleich von Testergebnissen spielen, um nur einige Beispiele zu nennen. Bei der Planung und Gestaltung des Einsatzes solcher Werkzeuge gilt es, die Testvorgehensweise, die Auswirkungen des implementierten agilen Softwareentwicklungslebenszyklus, die Einsatzmöglichkeiten dieser Testautomatisierungswerkzeuge und die Integration verschiedener anderer Werkzeuge mit diesen Testautomatisierungswerkzeugen (z. B. Testautomatisierung im Rahmen eines Continuous Integration Frameworks) zu berücksichtigen.

In einigen Fällen dient die Testautomatisierung direkt den Zielen einer Iteration, d.h. der Bereitstellung einer neuen Funktionalität. In anderen Fällen unterstützt die Testautomatisierung diese Ziele indirekt, z. B. durch die Reduzierung des Regressionsrisikos bei Änderungen am System. Wie im Foundation Lehrplan Agile Tester [AgileFoundationExt] erläutert, entscheiden sich einige Organisationen dafür, diese unterstützenden Testautomatisierungsaktivitäten an externe Teams außerhalb des Iterationsteams abzugeben. In derartigen Organisationen findet man zum Beispiel ein separates Team, welches die Erstellung und Wartung des Automatisierungsframeworks für Regressionstests als Service für mehrere Iterationsteams anbietet. Dieser Ansatz kann erfolgreich sein, wenn das externe Team den Iterationsteams nützliche Services zur Verfügung stellt, welche diese Iterationsteams dabei unterstützen, dass diese sich auf ihre unmittelbaren Iterationsziele konzentrieren können. Die Abhängigkeit von externen Teams kann das Commitment des Iterationsteams beeinflussen, da es teilweise die Kontrolle über sein Commitment an das externe Team überträgt.

Im Folgenden sind Überlegungen zur Testautomatisierung für die wichtigsten Teststrategien genannt, die in den Lehrplänen des ISTQB® Foundation Level, Advanced Level - Test Manager und Expert Level - Test Manager aufgeführt sind:

- **Analytisch:** Verhaltensgetriebene Entwicklung (Behavior-Driven Development, BDD) und abnahmetestgetriebene Entwicklung (Acceptance Test-Driven Development, ATDD) sind Verfahren, die im Rahmen einer analytischen Testvorgehensweise in einem agilen Kontext eingesetzt werden können, z. B. angewandt auf die Testautomatisierung. Mit BDD und ATDD können automatisierte Tests parallel zur (oder sogar vor der) Realisierung der User-Story erstellt werden.
- **Modellbasiert:** Modellbasiertes Testen des funktionalen Verhaltens kann die automatisierte Erstellung von Tests während der Realisierung der User-Story unterstützen und damit eine schnell zugängliche Quelle für effiziente Tests darstellen. Modellbasiertes Testen kann auch für die Erstellung von User-Storys

eingesetzt werden, da Modelle zum Testen von Anforderungen in ihren Abläufen und somit auch zur Unterstützung von statischen Reviews verwendet werden können. Modelle werden oft für das Testen von nicht-funktionalem Verhalten wie Zuverlässigkeit und Performance verwendet, die für viele Systeme wichtige Merkmale sind und die sich aus dem Gesamtsystem heraus entwickeln.

- **Methodisch:** Da es in agilen Projekten viele kurze Iterationen gibt, können automatisierte Test-Checklisten (mit allen Aktivitäten) als methodischer Ansatz für die effiziente Durchführung von einer stabilen Testfallmenge verwendet werden.
- **Prozesskonform:** Bei Projekten, die extern festgelegte Standards oder Vorschriften einhalten müssen, können diese Standards oder Vorschriften Einfluss darauf nehmen, wie automatisierte Tests eingesetzt oder wie automatisierte Testergebnisse erfasst werden. So müssen z. B. bei FDA-regulierten (Food & Drug Administration: US-amerikanische Bundesbehörde zur Überwachung von Nahrungs- und Arzneimitteln) und damit Hochrisiko-Projekten die automatisierten Tests und deren Ergebnisse zu den Anforderungen rückverfolgbar sein und die Ergebnisse müssen genügend Details für den Nachweis enthalten, dass der Test bestanden wurde.
- **Reaktiv** (oder heuristisch): Reaktives Testen spielt eine wichtige Rolle bei der Validierung im agilen Testen, während die meisten automatisierten Tests primär eine Rolle bei der Verifikation spielen. Da reaktive Testvorgehensweisen in erster Linie manuell sind (z. B. exploratives Testen, intuitive Testfallermittlung (Error Guessing)), führt ein erhöhter Anteil an automatisierter Testüberdeckung oft zu einem höheren Grad an manuellen, auf reaktiven Testvorgehensweisen basierenden Tests. Denn viele der Tests, die im Vorfeld vorbereitet werden können, werden automatisiert. Darüber hinaus können die verbleibenden manuellen Tests risikoreichere Bereiche abdecken.
- **Angeleitet** (oder beratend): Wenn die Testüberdeckung von externen Stakeholdern festgelegt wird und Testautomatisierung eingesetzt werden soll, ist die Fähigkeit des Testteams, auf die Anforderung zu reagieren, von Bedeutung. Daher sollten Testteams, die eine angeleitete (beratende) Testvorgehensweise verfolgen, sowohl die Zeit als auch die notwendigen Fähigkeiten berücksichtigen, die zur Erfüllung ihrer Aufgaben innerhalb der Iterationen erforderlich sind.
- **Regressionsvermeidend:** In agilen Projekten ist ein primäres Merkmal der regressionsvermeidenden Testvorgehensweise eine große, stabile und wachsende Menge an automatisierten Regressionstests. Eine adäquate Überdeckung, Wartbarkeit und effiziente Analyse der Ergebnisse sind entscheidend, insbesondere wenn die Anzahl der Regressionstests wächst. Anstatt sich auf eine ständig wachsende Anzahl von Regressionstests zu fokussieren, konzentriert sich eine erfolgreiche regressionsvermeidende Testvorgehensweise auf die kontinuierliche Verbesserung und auf das Refactoring der erstellten Tests.

3.2 Automatisierungsgrad

3.2.1 Verständnis des erforderlichen Grads der Testautomatisierung

Automatisierung ist ein wichtiges Element in agilen Projekten, da sie nicht nur die Testautomatisierung, sondern auch die Automatisierung des Bereitstellungs-/ Inbetriebnahmeprozesses (Deployment Process) umfasst (vgl. Kapitel 4). Continuous Deployment ist die automatisierte Bereitstellung bzw. Inbetriebnahme einer neuen Software-Version in die Produktionsumgebung. Das Continuous Deployment erfolgt in regelmäßigen und kurzen Abständen.

Da es sich beim Continuous Deployment um einen automatisierten Prozess handelt, müssen die automatisierten Tests ausreichend sein, um die erforderliche Qualität des Programmcodes aufrechterhalten zu können. Die bloße Ausführung von automatisierten Unittests ist nicht genügend, um eine ausreichende Testüberdeckung zu erzielen. Eine automatisierte Testsuite, die als Teil des Deployment-Prozesses ausgeführt wird, muss auch Tests auf Integrations- und Systemtestebene beinhalten. In der Praxis kann auch noch manuelles Testen erforderlich sein.

Im Folgenden sind einige Herausforderungen aufgeführt, denen man sich bei der Testautomatisierung in einem agilen Umfeld stellen muss:

- Umfang der Testsuite: Jede Iteration (außer Wartungsiterationen oder sog. Hardening Iterations / Stabilisierungsiterationen) realisiert zusätzliche Features. Um die Testsuite mit den zusätzlichen Funktionalitäten des Produkts in Einklang zu halten, muss das agile Team seine Testsuite innerhalb jeder Iteration erweitern. Das bedeutet, dass die Anzahl der Testfälle innerhalb der Testsuite von Iteration zu Iteration grundsätzlich steigt. Es erfordert sorgfältige und bewusste Maßnahmen zum Refactoring der Tests, um die Überdeckung zu erhöhen, ohne gleichzeitig den Umfang beträchtlich zu erhöhen. Auf jeden Fall werden der Aufwand und die Zeit, die für die Wartung, Vorbereitung und Ausführung der kompletten Testsuite erforderlich sind, mit der Zeit zunehmen.
- Entwicklungszeit für die Tests: Die Tests, die zur Überprüfung der neuen oder geänderten Funktionalität des Produkts erforderlich sind, müssen entworfen und realisiert werden. Dazu gehört die Erstellung bzw. Aktualisierung der notwendigen Testdaten und die Vorbereitung eventueller Updates der Testumgebung. Die Wartbarkeit der Tests wirkt sich auch auf die Entwicklungszeit aus.
- Testdurchführungszeit: Mit zunehmendem Umfang der Testsuite steigt auch der Zeitaufwand für die Durchführung der Tests.
- Verfügbarkeit der Mitarbeiter: Die Mitarbeiter, die für die Erstellung, Wartung und Ausführung der Testsuite benötigt werden, müssen bei jedem Deployment verfügbar sein. Es kann schwierig oder unmöglich sein, dies während des gesamten Projektverlaufs sicherzustellen, insbesondere während der Ferien, an

Wochenenden oder wenn Deployments außerhalb der regulären Arbeitszeiten stattfinden.

Eine Strategie zur Bewältigung dieser Herausforderung ist die Reduzierung der Testsuite, indem jeweils nur ein Teil der Tests ausgewählt, vorbereitet und durchgeführt wird, wobei die Auswahl der Tests auf Basis der Priorisierung und/oder Risikoanalyse getroffen wird. Diese Strategie hat den Nachteil, dass das Risiko erhöht wird, weil dadurch weniger Tests durchgeführt werden.

Die Automatisierung möglichst vieler Tests soll die Häufigkeit und/oder die Geschwindigkeit von Deployments erhöhen. Eine weitere Strategie, um mit der Häufigkeit und/oder der Geschwindigkeit von Deployments Schritt halten zu können (oder diese sogar zu erhöhen) besteht in der Automatisierung möglichst vieler Tests.

Testautomatisierung ist ein operatives Instrument, um im Projekt die Deployment-Geschwindigkeit zu halten oder zu erhöhen, und sie ist die Voraussetzung für ein Continuous Deployment. Um das richtige Maß an Testautomatisierung zu finden, das mit der Geschwindigkeit des Deployments Schritt halten kann, sollten die folgenden Vorteile und Einschränkungen analysiert und abgewogen werden:

Vorteile

- Testautomatisierung kann einen definierten und wiederholbaren Grad der Testüberdeckung für jeden Deployment-Zyklus sicherstellen.
- Testautomatisierung kann die Testdurchführungszeit verkürzen und dazu beitragen, die Geschwindigkeit des Deployments zu erhöhen.
- Testautomatisierung kann die Beschränkungen zur Erreichung einer höheren Deployment-Häufigkeit verringern.
- Continuous Deployment unterstützt ein kürzeres Time-to-Market sowie kürzere Feedback-Schleifen der Benutzer.
- Testautomatisierung kann häufiger eingesetzt werden, wodurch alle Tests mit jedem Build ausgeführt werden können. Dies ermöglicht eine stabile Baseline (mainline) im Rahmen der Continuous Integration, bei der die Software so oft wie möglich mit der Baseline zusammengeführt (merge) wird.

Einschränkungen

- Hohe Testentwicklungs- und Wartungsaufwände, die wiederum die Entwicklungsdauer verlängern und damit die Deployment-Häufigkeit verringern können.
- Tests auf Systemteststufe und insbesondere Last- oder Performancetests (und möglicherweise auch andere nicht-funktionale Tests) können, auch wenn sie automatisiert sind, eine lange Zeit für die Ausführung benötigen.

- Automatisierte Tests können an vielen Faktoren scheitern. Ein bestandener automatisierter Testfall ist möglicherweise nicht zuverlässig (falsch negatives Ergebnis). Ein fehlgeschlagener automatisierter Testfall kann aufgrund einer Fehlhandlung, die nicht mit der Produktqualität zusammenhängt, oder aufgrund eines falsch positiven Ergebnisses fehlschlagen.

Die Häufigkeit von Releases sollte den Bedürfnissen des Kunden entsprechen. Zu viele ausgelieferte Versionen in kurzer Zeit können dem Kunden eher missfallen als eine geringere Häufigkeit. Daher ist in Situationen, in denen es technisch möglich ist, die Häufigkeit der Deployments weiter zu erhöhen, dies aus Kundensicht möglicherweise nicht immer von zusätzlichem Nutzen.

4 Deployment und Delivery (Bereitstellung/Inbetriebnahme und Auslieferung) – 105 Minuten

Schlüsselbegriffe

Service-Virtualisierung, Continuous Testing (kontinuierliches Testen)

Lernziele für Deployment und Delivery

ATT-4.x (K1) Schlüsselbegriffe

4.1 Continuous Integration, Continuous Testing und Continuous Delivery

ATT-4.1.1 (K3) Continuous Integration anwenden können und deren Auswirkungen auf die Testaktivitäten zusammenfassen können

ATT-4.1.2 (K2) Rolle des Continuous Testing bei Continuous Delivery und Continuous Deployment verstehen können

4.2 Virtualisierung

ATT-4.2.1-1 (K2) Konzept der Service-Virtualisierung und deren Rolle in agilen Projekten verstehen können

ATT-4.2.1-2 (K2) Vorteile der Service-Virtualisierung verstehen können

4.1 Continuous Integration, Continuous Testing und Continuous Delivery

4.1.1 Continuous Integration und ihre Auswirkungen auf das Testen

Das Ziel der Continuous Integration (CI) ist es, ein schnelles Feedback zu geben, so dass, falls Fehlerzustände im Code eingefügt werden, diese gefunden und so schnell wie möglich behoben werden. Agile Tester sollten zum Design, zur Realisierung und zur Wartung eines effektiven und effizienten CI-Prozesses beitragen, nicht nur in Bezug auf die Erstellung und Wartung von automatisierten Tests, die in das CI-Framework passen, sondern auch in Bezug auf die Priorisierung der Tests, die notwendigen Umgebungen, die Konfigurationen, usw.

In einer idealen Welt mit CI werden nach Fertigstellung des Builds alle automatisierten Tests durchgeführt, um zu überprüfen, ob sich die Software weiterhin wie spezifiziert verhält und nicht durch die Codeänderungen beschädigt wurde. Es gibt jedoch zwei widersprüchliche Ziele:

1. Den CI-Prozess häufig ausführen, um ein sofortiges Feedback zum Code zu erhalten.
2. Den Code so gründlich wie möglich nach jedem Build überprüfen.

Wenn bei der Konzeption, Realisierung und Wartung der automatisierten Tests (wie im vorigen Kapitel erläutert) nicht genügend Sorgfalt aufgewendet wird, wird die Ausführung aller automatisierten Tests zu lange dauern, um den CI-Prozess mehrmals täglich abschließen zu können. Selbst bei sorgfältiger Testautomatisierung kann es vorkommen, dass die vollständige Ausführung aller automatisierten Tests über alle Teststufen den CI-Prozess übermäßig verlangsamen würde. Verschiedene Organisationen haben unterschiedliche Prioritäten und verschiedene Projekte benötigen unterschiedliche Lösungen, um die richtige Balance zwischen den oben genannten Zielen zu finden. Wenn zum Beispiel ein System stabil ist und sich weniger häufig ändert, dann sind möglicherweise weniger CI-Zyklen erforderlich. Wenn das System ständig aktualisiert wird, sind höchstwahrscheinlich mehr CI-Zyklen erforderlich.

Es gibt Lösungen, die beide Ziele unterstützen, sich gegenseitig ergänzen und parallel genutzt werden können.

Der erste Lösungsansatz besteht darin, die Testfälle unter Verwendung eines risikobasierten Testansatzes zu priorisieren, sodass die grundlegenden und wichtigsten Testfälle immer durchgeführt werden.

Der zweite Lösungsansatz besteht darin, die Verwendung verschiedener Testkonfigurationen im CI-Prozess für den Einsatz in unterschiedlichen CI-Zyklustypen zu ermöglichen. Für den täglichen Build- und Testprozess werden nur die grundlegenden Tests ausgeführt, die aufgrund der Priorisierung im Vorfeld ausgewählt wurden. Für den nächtlichen CI-Prozess werden eine größere Anzahl oder möglichst alle funktionalen Tests durchgeführt, die keine produktionsvergleichbare Umgebung benötigen. Vor der Freigabe für Produktion werden gründlichere funktionale und nicht-funktionale Tests in einer Vorproduktionsumgebung mit echten Benutzereingaben durchgeführt, einschließlich Integrationstests mit Datenbanken, verschiedenen Systemen und/oder Plattformen.

Die dritte Lösung besteht darin, durch die Verringerung der Anzahl an Tests der Benutzungsschnittstellen (User Interface, UI) die Testdurchführung zu beschleunigen. Normalerweise gibt es keine Zeitprobleme bei der Ausführung von Unit-/Integrationstests, da diese in der Regel sehr kurze Laufzeiten haben. Es kann aber zur Situation kommen, dass so viele Unittests existieren, dass sie während des CI-Prozesses nicht vollständig durchgeführt werden können. Die tatsächlichen Probleme mit der Laufzeit hängen meistens mit der Verwendung von End-to-End-UI-Testfällen als Teil des CI-Prozesses zusammen. Die Lösung besteht darin, den Umfang der API-, Befehlszeilen-, Daten- und Service-Layer-Tests sowie anderer Nicht-UI-Tests der Geschäftslogik zu erhöhen und die UI-Tests zu verringern, womit sich auch der Automatisierungsaufwand in der Testpyramide nach unten verschiebt. Dies stellt wartungsfreundlichere Tests sicher und reduziert auch die Testdurchführungszeit.

Die vierte Lösung kann eingesetzt werden, wenn die Testdurchführungen in einem CI-System sehr häufig sind und es unmöglich ist, alle Testfälle auszuführen. Basierend auf den Codeänderungen und Kenntnissen über die Ausführungspfade der

vorhandenen Testfälle kann ein Entwickler oder Tester nur die von den Änderungen betroffenen Testfälle auswählen und ausführen (d.h. Verwendung der Auswirkungsanalyse zur Auswahl von Testfällen). Da in der Regel nur ein kleiner Teil der gesamten Codebasis in einem kurzen Zyklus verändert wird, müssen relativ wenige Testfälle ausgeführt werden. Allerdings können dabei jedoch wesentliche Regressionsfehler übersehen werden.

Eine fünfte Lösung besteht darin, die Testsuite in vergleichbare Mengen aufzuteilen und diese parallel auf mehreren Umgebungen (Agenten, Build-Farm, Cloud) laufen zu lassen. Dies wird sehr häufig in Unternehmen mit CI eingesetzt, da diese ohnehin eine große Build-Server-Kapazität benötigen.

Die Continuous Integration arbeitet auf einer Vielzahl von Technologieplattformen. So wie die Entwicklungs- und Installationswerkzeuge haben sich auch die CI-Produkte in Richtung Cloud entwickelt. Während die meisten CI-Produkte jedoch noch immer für den Download und die Ausführung in lokalen Umgebungen konzipiert sind, hat die Cloud eine neue Art von Produkten geschaffen, die CI-Services auf gehosteten Plattformen bereitstellen, mit deren Nutzung die Teams schnell beginnen können. So können Teams die überflüssigen Kosten und den Zeitaufwand für das Erstellen einer neuen Umgebung, den Download, Installieren und Konfigurieren der CI-Software vermeiden. Durch den Wechsel in die Cloud kann das Team die Konfiguration durchführen und direkt mit der Arbeit beginnen. Wenn möglich ist die Cloud eine flexible Lösung, um den Test Build und die Testdurchführung bei Bedarf zu beschleunigen.

Aktuelle CI-Tools unterstützen nicht nur Continuous Integration, sondern auch Continuous Delivery (kontinuierliche Auslieferung) und Continuous Deployment. Die Durchführung automatisierter Tests in einer Umgebung, die die Produktion nicht vollständig widerspiegelt, kann zu falsch negativen Ergebnissen führen, jedoch kann das Klonen der Produktionsumgebung hohe Kosten verursachen. Lösungen sind unter anderem die Verwendung von cloudbasierten Testumgebungen, die Produktionsumgebungen nach Bedarf replizieren, oder die Einrichtung einer Testumgebung, die eine verkleinerte aber realistische Version der Produktion darstellt.

Gute CI-Systeme sollten in der Lage sein, sich auch in komplexeren Umgebungen und auf verschiedenen Plattformen automatisch zu installieren. Die Aufgabe der Tester ist es zu planen, welche Testfälle einbezogen werden sollen, und zu priorisieren, welche auf einigen oder allen Plattformen ausgeführt werden müssen, um eine gute Überdeckung zu gewährleisten. Dabei sind die Testfälle so zu entwerfen, dass die Software in einer produktionsnahen Umgebung effizient validiert werden kann.

4.1.2 Rolle des Continuous Testing bei Continuous Delivery und Continuous Deployment

Continuous Testing ist ein Ansatz, bei dem früh, häufig und überall getestet und automatisiert wird, um so schnell wie möglich Feedback zu den mit einem Release-

Kandidaten verbundenen Geschäftsrisiken zu erhalten. Beim Continuous Testing löst eine Änderung am System die erforderlichen Tests (d.h. diejenigen Tests, welche die Änderung abdecken) automatisch aus und gibt dem Entwickler umgehendes Feedback. Continuous Testing kann in verschiedenen Situationen angewendet werden, z. B. in einer integrierten Entwicklungsumgebung (IDE), in der Continuous Integration (CI), beim Continuous Delivery (CD), beim Continuous Deployment, usw. Das Konzept ist jedoch immer dasselbe, nämlich dass die Tests so früh wie möglich automatisiert durchgeführt werden.

Continuous Delivery erfordert Continuous Integration. Continuous Delivery erweitert Continuous Integration, indem alle Codeänderungen nach der Build-Phase in einer Test- oder Vorproduktionsumgebung und/oder einer produktionsvergleichbaren Umgebung installiert werden. Dieser Staging-Prozess ermöglicht die Durchführung von funktionalen Tests mit realen Benutzereingaben und nicht-funktionalen Tests wie Last-, Stress-, Performanz- und Portabilitätstests. Da jede eingebettete Änderung mittels vollständiger Automatisierung an die Staging-Umgebung geliefert wird, kann das agile Team darauf vertrauen, dass das System mit einem Knopfdruck in die Produktion ausgerollt werden kann, sobald die Definition of Done erreicht ist.

Continuous Deployment geht noch einen Schritt weiter als Continuous Delivery, indem die Änderung automatisch in die Produktion übernommen wird. Continuous Deployment zielt darauf ab, die Zeitspanne zwischen der Entwicklungsaufgabe, den Code zu schreiben, und der Nutzung dieses Codes durch reale Benutzer in der Produktion zu minimieren. Für weitere Informationen siehe [Farley].

4.2 Service-Virtualisierung

Service-Virtualisierung (service virtualization, Dienstvirtualisierung) bezieht sich auf den Prozess der Erstellung eines gemeinsam nutzbaren Testdienstes (eines virtuellen Services), der das Verhalten, die Daten und die Leistung eines angebundenen Systems oder Dienstes entsprechend simuliert. Dieser virtuelle Service ermöglicht es Entwicklungs- und Testteams, ihre Aufgaben auszuführen auch wenn sich der eigentliche Service noch in der Entwicklung befindet oder nicht verfügbar ist.

Das Testen in einem frühen Stadium des Softwareentwicklungslebenszyklus ist bei den heutigen stark vernetzten und voneinander abhängigen Entwicklungsteams und Systemen nur schwierig zu erreichen. Durch die Entkopplung von Teams und Systemen mittels Service-Virtualisierung können Teams ihre Software früher im Softwareentwicklungslebenszyklus mit realistischeren Anwendungsfällen (Use-Cases) und unter realistischerer Last testen.

Während Stubs, Treiber und Mocks wertvoll für die Möglichkeit eines frühen Testens sind, sind manuell erstellte Stubs in der Regel zustandslos und geben nur einfache Antworten mit festen Antwortzeiten auf Anfragen. Virtuelle Services können zustandsbehaftete Transaktionen unterstützen und den Kontext dynamischer Elemente (z. B. Session-/Kunden-IDs, Datums- und Zeitangaben) verwalten sowie

variable Antwortzeiten haben, die den Nachrichtenfluss durch mehrere Systeme abbilden.

Virtuelle Services werden mit Hilfe von Service-Virtualisierungswerkzeugen erstellt, häufig mit einer der folgenden Methoden:

- Durch die Interpretation von Daten: XML-Datei, historische Daten aus Server-Protokollen oder einfach nur ein Tabellenblatt mit Beispieldaten.
- Durch die Überwachung des Netzwerkverkehrs: Das Ausführen des SUT (System under Test) löst ein entsprechendes Verhalten des abhängigen Systems aus, welches durch das Service-Virtualisierungswerkzeug erfasst und nachgebildet wird.
- Durch die Aufzeichnung über Agenten: Server-seitige oder interne Logik ist möglicherweise nicht in der Nachrichtenkommunikation sichtbar, so dass erzwungen werden muss relevante Daten und Nachrichten vom abhängigen Server zu senden, um als virtueller Service nachgebildet zu werden.

Ist keiner der oben genannten Punkte anwendbar, so muss der virtuelle Service innerhalb des Projektteams auf der Grundlage des entsprechenden Kommunikationsprotokolls erstellt werden.

Zu den Vorteilen der Service-Virtualisierung gehören:

- Parallele Entwicklungs- und Testaktivitäten für den zu entwickelnden Dienst
- Frühere Tests von Diensten und APIs
- Frühere Bereitstellung der Testdaten
- Parallele Kompilierung, kontinuierliche Integration und Testautomatisierung
- Früheres Aufdecken von Fehlerzuständen
- Verringern einer Überbeanspruchung von gemeinsam genutzten Ressourcen (kommerzielle Standardsoftware (Commercial Off-The-Shelf, COTS), Mainframe, Data Warehouse)
- Reduzierte Testkosten durch Reduzierung der Investitionen in die Infrastruktur
- Ermöglichung eines frühen nicht-funktionalen Testens des SUT
- Vereinfachung des Testumgebungsmanagements
- Geringerer Wartungsaufwand für Testumgebungen (keine Wartung der Middleware erforderlich)
- Geringeres Risiko für das Datenmanagement (was bei der GDPR[1] Compliance hilft)

[1] General Data Protection Regulation (GDPR) ist der aktuelle rechtliche Rahmen der Europäischen Union für eine Allgemeine Datenschutz-Verordnung.

Zu beachten ist, dass ein virtueller Service nicht alle Funktionen und Daten des aktuellen Dienstes enthalten muss, sondern nur die Teile, die zum Testen des SUT benötigt werden.

Die Einführung einer Service-Virtualisierung kann ein komplexes und potenziell eher teures Unterfangen werden. Die Einführung eines Service-Virtualisierungswerkzeuges sollte ähnlich behandelt werden wie die Einführung jedes neuen Testwerkzeuges im Team bzw. in einer Organisation. Dieses Thema wird im ISTQB® Foundation Syllabus und im ISTQB® Advanced Test Manager Syllabus behandelt.

5 Referenzen

5.1 Normen/Standards

[ISO 29119-5], ISO/IEC/IEEE 29119-5 Software and systems engineering -- Software testing -- Part 5: Keyword-Driven Testing

5.2 ISTQB-Dokumente

ISTQB®-Glossar

[Foundation] ISTQB® Foundation Level Lehrplan 2020 (V3.1)

[AgileFoundationExt] ISTQB-AT Foundation Level Specialist Lehrplan Agile Tester 2017

[AdvancedTestAutomationEngineer] ISTQB-TAE Advanced Level Lehrplan Testautomatisierungsentwickler 2019

[ISTQB_ATT_OVIEW] Advanced Level Agile Technical Tester Overview document (ISTQB-CTAL-ATT_BO-LO-Overview_V1.1.pdf) 2020

5.3 Bücher und Artikel

[Adzic09] Adzic, Gojko. “Bridging the Communication Gap” Neuri Ltd, 2009

[Adzic11] Adzic, Gojko. “Specification by Example” Manning, 2011

[Anderson01] Lorin W. Anderson, David R. Krathwohl (Hrsg.) “A Taxonomy for Learning, Teaching and Assessing: A Revision of Bloom’s Taxonomy of Educational Objectives”, Allyn & Bacon, 2001, ISBN 978-0801319037

[Beck02] Kent Beck, “Test Driven Development: By Example”, 2002

[Beck99] Beck, Kent. “Extreme Programming Explained” Addison Wesley, 1999

[Carkenord08] Barbara A. Carkenord, “Seven Steps to Mastering Business Analysis”, J. Ross Publishing, 2008.

[Cohn 09] Mike Cohn: Succeeding with Agile: Software Development Using Scrum. Addison-Wesley Professional, 2009

[Crispin08] Crispin, L. and Gregory, J. (2008) Agile Testing: A Practical Guide for Testers and Agile Teams. Crawfordsville: Addison-Wesley Professional

[Elfriede99] Dustin Elfriede, Automated Software Testing: Introduction, Management, and Performance: Introduction, Management, and Performance, Addison-Wesley Professional, 1999

[Evans03] Eric Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software”, 2003

[Fewster12] Mark Fewster, Dorothy Graham, Experiences of Test Automation: Case Studies of Software Test Automation, Addison-Wesley Professional, 2012

[Fowler/Parsons 10], Martin Fowler, Rebecca Parsons, Domain-Specific Languages, Addison-Wesley Signature, Series, 2010

[Hendrickson13] Elisabeth Hendrickson, “Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing”, Pragmatic Bookshelf, 2013

[Jeffries00] Ron Jeffries, Ann Anderson, and Chet Hendrickson, “Extreme Programming Installed,” Addison-Wesley Professional, 2000

[Jorgensen13] Paul C. Jorgensen, Software Testing: A Craftsman’s Approach, Auerbach Publications; 4th edition, 2013

[Linz14] Tilo Linz, Testing in Scrum, A Guide for Software Quality Assurance in the Agile World, Rocky Nook, 2014

[Meszaros07] Gerard Meszaros, “xUnit Test Patterns: Refactoring Test Code”, Addison-Wesley, 1st Edition, Apress, 2007

[Michelsen12] John Michelsen and Jason English, “Service Virtualization: Reality is Overrated”, Apress, 1st Edition, 2012

[Osherove09] Roy Osherove, “The Art of Unit Testing”, 2009

[Paskal15] Greg Paskal, “Test Automation in the Real World: Practical Lessons for Automated Testing”, MissionWares, 2015

[Smart15]; Smart, John Ferguson; "BDD in action", Manning, 2015

[Wein89] Weinberg, Gerald & Gause, Donald. “Exploring Requirements: Quality Before Design” Dorset House, 1989.

[Whittaker09] James A Whittaker, “Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design”, Addison-Wesley Professional, 2009

[Wiegers02] Karl Wiegers, “Peer Reviews in Software: A Practical Guide (Paperback)”, 2002.

Agile Terminologie

Schlüsselbegriffe, die im ISTQB®-Glossar zu finden sind, werden am Anfang jedes Kapitels genannt. Für gebräuchliche agile Begriffe haben wir uns auf die folgende Internetquelle mit hoher Akzeptanz gestützt, die Definitionen liefert: https://www.agilealliance.org/agile101/agile-glossary/
Im Falle widersprüchlicher Definitionen ist das ISTQB®-Glossar die führende Quelle.

Weitere Referenzen

Die folgenden Verweise weisen auf Informationen hin, die im Internet und anderswo verfügbar sind. Auch wenn diese Referenzen zum Zeitpunkt der Veröffentlichung dieses Lehrplans überprüft wurden, kann das ISTQB® nicht verantwortlich gemacht werden, wenn die Referenzen nicht mehr verfügbar sind.

[Cohn09] The Forgotten Layer of the Test Automation Pyramid, https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

[CyclomaticComplexity] https://en.wikipedia.org/wiki/Cyclomatic_complexity

[Farley] http://www.davefarley.net/?cat=5

[DSL] https://en.wikipedia.org/wiki/Domain-specific_language

[Fowler07] https://martinfowler.com/articles/mocksArentStubs.html

[Fowler04] Fowler, Martin. https://martinfowler.com/bliki/SpecificationByExample.html

[Fowler03] Martin Fowler, https://martinfowler.com/bliki/TechnicalDebt.html

[Gherkin] https://docs.cucumber.io/gherkin/

[INVEST] Bill Wake, “INVEST in Good Stories, and SMART Tasks”, http://xp123.com/articles/investin-good-stories-and-smart-tasks/

[IQBBA] International Qualifications Board for Business Analysts, http://www.iqbba.org/

[IREB] International Requirements Engineering Board, https://www.ireb.org/en

[IIBA] International Institute of Business Analysts (IIBA), https://www.iiba.org/

[Marick01] Marick, Brian, http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1

[Marick03] Marick, Brian. http://www.exampler.com/old-blog/2003/08/22.1.html

[North06] Dan North, “Introducing BDD”, blog post, 2006

[TESTDOUBLES] Wojciech Bulaty, Bill Wake, “Stubbing, Mocking and Service Virtualization Differences for Test and Development Teams”, https://www.infoq.com/articles/stubbing-mockingservice-virtualization-differences

[ToolList] Test tool review, information platform on the international market of software testing tools, www.testtoolreview.de/en/

[xUnit] https://en.wikipedia.org/wiki/XUnit, https://en.wikipedia.org/wiki/Unit_testing

6 Anhang

6.1 Glossar zu Agile Technical Tester spezifischen Fachbegriffen

Glossar Begriff	**Definition**
Persona	Ein fiktiver Charakter, der eine bestimmte Art von Benutzern und deren Interaktion mit dem System repräsentiert.
Storyboard	Eine visuelle Darstellung des Systems, in der User-Storys im Kontext dargestellt werden, um die Geschäftsprozesse zu verstehen.
Story-Mapping	Eine Technik, die User-Storys auf zwei Dimensionen anordnet, wobei die horizontale Achse ihre Ausführungsreihenfolge und die vertikale Achse die Komplexität des implementierten Produkts repräsentiert.